National Library of Canada
Canadian Theses Service
Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Service des thèses canadiennes

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

NL-339 (r. 88/04) c

Canada

UNIVERSITY OF ALBERTA

ADACC - A SYNTHESIS TOOL FOR THE DESIGN OF ASYNCHRONOUS FINITE STATE MACHINES

BY

TREVOR C. MAY

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE.

DEPARTMENT OF ELECTRICAL ENGINEERING

EDMONTON, ALBERTA

SPRING, 1991

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-66690-0

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Trevor C. May

TITLE OF THESIS:

ADACC - A Synthesis Tool For The Design Of Asynchronous Finite State Machines

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1991

PERMISSION IS HEREBY GRANTED TO THE UNIVERSITY OF ALBERTA LIBRARY TO REPRODUCE SINGLE COPIES OF THIS THESIS AND TO LEND OR SELL SUCH COPIES FOR PRIVATE, SCHOLARLY OR SCIENTIFIC RESEARCH PURPOSES ONLY.

THE AUTHOR RESERVES OTHER PUBLICATION RIGHTS, AND NEITHER THE THESIS NOR EXTENSIVE EXTRACTS FROM IT MAY BE PRINTED OR OTHERWISE REPRODUCED WITHOUT THE AUTHOR'S WRITTEN PERMISSION.

Trevor May

(Student's Signature)

10915 85 Ave
Edmonton, Alberta
Canada
T6G 0W3

Date: Dec 28, 1990

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

THE UNDERSIGNED CERTIFY THAT THEY HAVE READ, AND RECOMMEND TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH FOR ACCEPTANCE, A THESIS ENTITLED: ADACC - A Synthesis Tool For The Design of Asynchronous Finite State Machines

SUBMITTED BY: Trevor C. May

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE.

Emil F. Girczyc, co-supervisor

Jack T. Mowchenko, co-supervisor

Duane A. Szafron

Bruce F. Cockburn

Date: Dec. 20. 1990

Dedication

The author would like to dedicate this thesis to the following people and groups:

1. Emil Girczyc, for allowing enough rope for academic freedom. Unfortunately this rope was also long enough for other purposes...
2. IDACOM Electronics, for giving me a proper engineering education.
3. C. D. Shaw, for intelligent, if often bizarre, conversation.
4. 'Rebecca's Bath', for taking up a good chunk of my free time, and most of my spare money.
5. Lisa Buchner, for providing emotional support and for taking up the balance of my free time and money.
6. Chuck and Phyllis May, for understanding when I had no time left for them.

Abstract

In this thesis ADACC is presented. ADACC is a synthesis tool for the automatic generation of Asynchronous Finite State Machine (AFSM) circuitry from high-level descriptions. ADACC starts with an ASCII description of the user's state machine and generates a gate-level netlist of circuitry to implement the state machine.

Novel features of ADACC include:

1 A state assignment system that uses simulated annealing and an algorithmic clean up procedure. This system generates race-free state assignments that avoid jeopardizing the user's timing requirements by not placing transition states into transitions that are required to be faster than average.

2 Removal of essential hazard effects with a rule based timing optimizer. This optimizer adjusts the circuit timing to ensure that essential hazards are hidden, and to meet most of the user's timing constraints. This is quite different from traditional approaches of essential hazard removal which simply place delays into circuit feedback paths.

ADACC was tested by using it to generate several AFSM circuits, which were then simulated to verify correct operation. It was found that ADACC produced circuitry that did not exhibit any circuit hazards, and operated as the user intended. Unfortunately, ADACC does not meet all user timing constraints. This problem is discussed in the thesis and a possible solution is presented.

Acknowledgements

The author would like to thank the following people and groups for providing support throughout this research:

1. L. Klassen, for supplying G++.

2 Norman Jantz, for providing the technical support required to develop ADACC.

3. The Natural Sciences and Engineering Research Council of Canada (NSERC) for providing much needed research funds.

Table of Contents

Chapter 1: Introduction 1
Thesis Overview 2
Chapter 2: Background Research 4
The Classical Method of AFSM Design 4
Critical Race Hazards 5
Combinational Hazards 5
Static Hazards 6
Dynamic Hazards 7
Essential Hazards 7
Elimination of Hazards 9
Solving Critical Race Hazards 9
Removing Combinational Hazards 10
Removing the Effects of Essential Hazards 10
Classical AFSM Theory Used in ADACC 11
Speed-Independent Circuits 12
Synthesis of Speed-Independent Circuits 13
Disadvantages of Speed-Independent Circuits 14
Rule-Based Logic Synthesis 15
General Rule-Based Systems 15
LSS 16
SOCRATES 17
Logic Synthesis Theory Used in ADACC 18
Chapter 3: High-Level Overview of ADACC 19
Introduction 19
User State Machine Input 19
State Assignment 20
Equation Generation 20
Generation of Output Equations 21
Generation of Next State Equations 21
Combinational Hazard Remover 23
Essential Hazard Detector 23
Timing Restriction Generator 24
Default Transition Time Restrictions 25
Transition Time Restrictions 26
Essential Hazard Timing Restrictions 27
Rule-Based Timing Optimization 28
Use of Rule-Based Optimization to Hide Essential Hazard Effects 28

Requirements of Rule-Based Optimization System 28
Selecting The Rule Base 29
Chapter 4: Implementation of ADACC 33
Overview 33
Important Objects Used in ADACC 33
Data Class 35
Graph Class 35
Children of the Graph Class 36
Rule Class 37
State Assignment Implementation 39
Overview 39
State Splitting Pre-processor 40
Initial Assignment with Simulated Annealing 42
Annealing Cost Function 45
Annealing Temperature Scales 46
Implementation of Algorithmic Cleanup Procedure 48
Removal of Combinational Hazards 51
Detection of Essential Hazards 53
Implementation of Rule-Based Timing Optimization System 55
Technology Mapper 57
Creation of Generic Circuitry 57
Technology Dependent Gate Library 58
Mapping Rules 58
Rule Application 60
Rule Matching Mechanism 60
Rule Application Mechanism 60
Timing Optimizer 61
Rule Maintenance System 62
Conflict Resolver 66
Timing Analyser 68
Post Timing Optimization 70
Chapter 5: State Assignment Test Results 71
Chapter 6: ADACC Test Results 75
Overview 75
Test Machine 1 - Lab1 of EE535 75
Test Machine 2 - VME Bus Arbitrator 87
Test Machine 3 - HDLC Protocol Serial Bit Stuffer 95
Test Machine 4 - FM (Single Density) Floppy Disk Data Separator 102
Chapter 7: Conclusions and Additional Research 109

Additional Research 110
State Assignment For Reduced Equation Complexity 110
Elimination of Fundamental Mode Assumption 110
Optimization Performance 110
Optimization for Reduced Area 111
Timing Optimization to Increase the Speed of Transition and Shared States 111
Design For Testability 112
Chapter 8: References 113
Appendix A: Syntax of ADACC Parsers 117
Appendix B: Synthesis Gate Library 124
Appendix C: Important Methods of the Data and Graph Class 125
Appendix D: Technology Mapping and Timing Optimization Rules 127
Appendix E: Input to ADACC for VME bus Arbitrator 132

List of Tables

Table 5.1. Assignment performance with minimum number of state variables 72
Table 5.2. Assignment performance with one additional state variable 73
Table 6.1. Essential hazards detected by ADACC 77

List of Figures

Figure 2.1. Static 1 hazard 6
Figure 2.2. Static 0 hazard 6
Figure 2.3. Low-to-high dynamic hazard 7
Figure 2.4. High-to-low dynamic hazard 7
Figure 2.5. Example flow table with essential hazard 8
Figure 2.6. Example flow table without essential hazard 8
Figure 3.1. Example state machine 21
Figure 3.2. Example state machine 22
Figure 3.3. Primary and feedback paths 25
Figure 3.4. Secondary path 27
Figure 3.5. Boolean algebra based topographical optimization rules 30
Figure 3.6. Gate delay optimization rules 31
Figure 4.1.1. Block diagram of ADACC 34
Figure 4.2.1. Typical cone of influence 37
Figure 4.2.2. Pointer structure of a typical rule 39
Figure 4.3.1. Top state splitting example 41
Figure 4.3.2. Bottom state splitting example 42
Figure 4.3.3. Pseudo-code for simulated annealing 45
Figure 4.3.4. Pseudo-code for 'cleanup()' 49
Figure 4.3.5. Pseudo-code for 'complete_arc()' 50
Figure 4.3.6. Example of 'complete_arc()' operation 51
Figure 4.4.1. Pseudo-code for 'cover_hazards()' 52
Figure 4.4.2. Pseudo-code for 'find_cover_term()' 53
Figure 4.5.1. Pseudo-code for 'find_hazards()' 54
Figure 4.6.1. Example of a circuit created from a Boolean equation 58
Figure 4.6.2. Library gates used in mapping rules 59
Figure 4.6.3. Pseudo-code for 'apply()' 60
Figure 4.6.4. Pseudo-code for timing optimization 62
Figure 4.6.5. Pseudo-code for 'delete_outdated_rules()' 63
Figure 4.6.6. Pseudo-code for 'add_new_rules()' 65
Figure 4.6.7. Pseudo-code for 'distribute_rule()' 66
Figure 4.6.8. Pseudo-code for 'resolve_conflict()' 67
Figure 4.6.9. Pseudo-code for 'global_evaluate_rule()' 68
Figure 4.6.10. Pseudo-code for 'check_timing()' 69
Figure 4.6.11. Pseudo-code for 'find_req_timing()' 70
Figure 6.1. Starting FSM for inhibited-toggle flip-flop 76
Figure 6.2. FSM for inhibited-toggle flip-flop after assignment 76

Figure 6.3. Next state and output equations 76
Figure 6.4a. Schematic of variable 'Q1+' after technology mapping 78
Figure 6.4b. Schematic of variable 'Q2+' after technology mapping 79
Figure 6.4c. Schematic of variable 'out' after technology mapping 80
Figure 6.5. Test results of SILOS simulation of original circuit 81
Figure 6.6. Schematic of variable 'Q1+' showing additional inverters 82
Figure 6.7. Test results of SILOS simulation showing essential hazard 83
Figure 6.8. Rules used to eliminate essential hazard 83
Figure 6.9a. Schematic of state variable 'Q1+' after optimization 85
Figure 6.9b. Schematic of state variable 'Q2+' after optimization 86
Figure 6.10. Test results of final circuit using SILOS 87
Figure 6.11. Initial VME arbiter state machine 89
Figure 6.12. Next state equations for FSM in Figure 6.11 90
Figure 6.13. Rules used in optimization 91
Figure 6.14. VME arbiter test case 1 92
Figure 6.15. VME arbiter test case 2 93
Figure 6.16. VME arbiter test case 3 94
Figure 6.17. Initial FSM for bit stuffer 96
Figure 6.18. FSM for bit stuffer after state assignment 97
Figure 6.19. Topographical rules used in optimization 98
Figure 6.20. Bit stuffer test case 1 99
Figure 6.21. Bit stuffer test case 2 99
Figure 6.22. Bit stuffer test case 3 100
Figure 6.23. Bit stuffer test case 4 100
Figure 6.24. FM encoding of a simple bit stream 102
Figure 6.25. Original FSM to implement FM data separator 104
Figure 6.26. FSM to implement FM data separator after assignment 105
Figure 6.27. FM data separator test case 1 106
Figure 6.28. FM data separator test case 2 107
Figure 6.29. FM data separator test case 3 107

1. Introduction

Asynchronous circuits are an important part of a digital designer's arsenal of tools. What separates these circuits from their synchronous counterparts is that they do not need a clock to be able to operate. That is, they need no external synchronization to ensure correct timing of the various circuit components. This gives asynchronous circuits some advantages over synchronous circuits.

The first advantage can be seen by observing that the maximum speed at which a synchronous circuit can operate is fixed by the slowest part (or critical path) of the circuit. This is because the circuit clock must be chosen to guarantee correct operation of all parts of the circuit, and is therefore chosen to reflect the speed of the critical path. Asynchronous circuits are not bound in this manner and therefore all circuit components can operate at their maximum speed. This can give a great speed improvement for the overall design.

The second advantage can be seen by noting the problem of clock skew in large I.C. designs and board level systems. Due to the physical layout of these designs, the clock signal used in one portion of the design may be drastically out of phase with the same clock signal in another portion. This violates a basic assumption of synchronous circuits, and could cause erroneous operation. This problem can be avoided by the use of asynchronous circuits to provide a clock-free interface between synchronous circuit blocks.

The third advantage is that asynchronous circuits can change states as soon as an input is detected. This is in contrast to synchronous designs that can only change states after a clock pulse. Here, synchronous circuits can be slower than similar asynchronous designs if the inputs are not well synchronized to the system clock.

The main disadvantage of asynchronous circuits is that they are difficult to design. The classical method of asynchronous finite state machine (AFSM) design depends on the elimination of several circuit hazards to guarantee correct circuit operation [Roth79]. These hazards range from simple combina-

tional hazards to essential hazards. The elimination of the effects of essential hazards must be done with careful control of the circuit timing, which can be very tedious for non-trivial circuit designs. Therefore, a synthesis tool that can aid the design of asynchronous circuits would be of great value.

There are two widely used varieties of asynchronous circuits. The first is asynchronous finite state machines (AFSMs), which are a variation of clocked synchronous finite state machines. As stated previously, the theory of operation of these types of asynchronous circuits is sometimes called classical AFSM theory. Because this class is an extension of synchronous FSMs, timing (and other) constraints must be placed on the circuit in order to guarantee correct operation.

The second variation of asynchronous circuits is called self-timed or speed-independent circuits (see Chapter 7 of [Mead80]). This class of circuits differs from AFSMs in that it is possible to guarantee correct operation without worrying about relative delays of the circuit components, whereas the timing of AFSMs must be carefully controlled to guarantee correct operation.

Currently, there is much research activity in the synthesis of self-timed asynchronous circuits [Chu86], [Borr87], [Borr88], [Meng89]. However at this time there is very little activity in the synthesis of AFSMs. In addition, the advent of logic synthesis presents a tool to help control the timing of any digital circuit. This research focuses on the investigation of automatic synthesis of AFSMs using rule based timing optimization to help control circuit timing. The main objective of this thesis is to show that a design tool can be created to synthesize working AFSMs using classical asynchronous circuit theory and rule based timing optimization to help adjust the circuit timing to guarantee correct circuit operation. As a bonus, timing optimization can also be used to help meet any user-specified timing constraints.

1.1. Thesis Overview

The result of this research is a synthesis tool called ADACC. In this thesis, the theory of operation, the implementation, and the testing of ADACC is described. Several test cases are shown in which

ADACC is used to synthesize a variety of circuits. The correct operation of these circuits is demonstrated using simulation, and ADACC's performance with these circuits is discussed.

Interesting features of ADACC are:

1. Automatic State Assignment using simulated annealing.
2. Automatic static hazard detection/correction.
3. Automatic essential hazard detection.
4. Essential hazard timing restrictions and most user timing restrictions are met with the help of rule-based timing optimizer.

ADACC takes input in the form of a state machine entry language, the syntax of which is in Appendix A, and produces output in the form of a netlist that describes the final circuit. This netlist is readable by a number of CAD tools such as the SILOS logic simulator [Simu88].

Chapter 2 of this thesis contains the background information used in ADACC. Chapter 3 presents an overview of ADACC and the theory of its operation. Chapter 4 contains the implementation details of ADACC. Chapters 5 and 6 contain the results of various experiments performed using ADACC. Chapter 7 includes the conclusions and some ideas for additional research.

2. Background Research

This thesis encompasses a relatively wide range of disciplines. This chapter surveys these disciplines giving background information to help the reader understand the content of this thesis.

This chapter is divided into 4 sections. The first section describes the classical method of AFSM design, including descriptions of circuit hazards and what can be done to eliminate these hazards. The second section overviews the theory behind speed-independent circuits, while the third section describes current research in the synthesis of speed-independent circuits. The fourth section contains an overview of rule-based logic synthesis.

2.1. The Classical Method of AFSM Design

Modern analysis and design of asynchronous circuits has its roots in a paper by Unger [Unge59]. In this paper, Unger describes the Feedback Delay Model for analysing and designing asynchronous circuits. This model, and Unger's theories of its operation, have been well explained by McCluskey [Mccl65], and is now considered to be textbook material [Unge69], [Lewi74], [Roth79], [Mccl86].

The design method for AFSMs is nearly the same as that of synchronous FSMs. The design method for synchronous FSMs is briefly described below:

1. A flow table is created that describes required circuit functionality.
2. Unique state vectors are assigned to each row in the table.
3. Combinational circuits are designed to implement each variable of the state vectors, as well as circuit outputs.
4. State variable outputs are used as inputs to clocked memory elements.
5. Outputs of these clocked memory elements are used as feedback in the circuit.

The main difference between synchronous FSM design and AFSM design is that, in AFSM design, the state variables are applied *directly* to the inputs of the combinational circuits, rather than running them through a clocked memory element. Because the memory element is removed from the feedback path, the design of asynchronous finite state machines must take into account several types of

circuit malfunctions, or *hazards*. These hazards are divided into three groups:

1. Critical Race Hazards
2. Combinational Hazards
3. Essential Hazards

Each of these hazard types must be either removed or hidden in any AFSM designed with the classical method in order to guarantee correct functionality. These hazards are described in more detail in the following sections.

2.1.1. Critical Race Hazards

When two or more state variable signals are changing at the same time, they are said to be *racing*. Races can occur between two or more state variables during a state change, or between two or more inputs during a change in inputs. When the final state of the circuit is dependent on which signal arrives at its destination first, these races are known as *critical* races.

Critical races can cause problems in the following situation. Let's assume that an AFSM has a transition from the state [01] to the state [10]. During this transition, if the first state variable is faster than the second, the AFSM may pass through the state [11] while on its way to state [10]. This could cause the machine to halt in state [11], rather than proceeding to state [10], which is a circuit malfunction. If this occurs, then the state variables are in a critical race. Note that if the circuit eventually proceeds to [10], then the race did not cause any permanent malfunction, and is therefore known as a *non-critical* race.

2.1.2. Combinational Hazards

Combinational hazards are circuit malfunctions or glitches that can appear on the outputs of simple combinational circuits. In asynchronous circuits these hazards have the potential for causing glitches on feedback variables that can make the circuit halt in an incorrect state, or start the circuit oscillating between two or more states. These hazards are divided into two classes, based on the behaviour of the

circuit output when the hazard occurs. These classes are called Static and Dynamic hazards respectively.

2.1.2.1. Static Hazards

Static hazards are false transient outputs of combinational circuits that appear when the circuit output is intended to be constant. That is, a network exhibits a static hazard if, in response to some input change, a momentary pulse on an output appears when that output should stay at a constant level.

Static hazards can be divided into two subclasses:

1. Static 1 hazards
2. Static 0 hazards

Static 1 hazards occur when a logic '0' transient appears on an output that is supposed to stay at a constant '1' value, and static 0 hazards occur when a '1' transient appears on an output that is supposed to stay at a constant '0' value. An example of a static 1 hazard is shown in Figure 2.1, and an example of a static 0 hazard is shown in Figure 2.2.

Figure 2.1. Static 1 hazard [Roth79]. Signal should stay at a constant 1 level.

Figure 2.2. Static 0 hazard [Roth79]. Signal should stay at a constant 0 level.

2.1.2.2. Dynamic Hazards

Dynamic hazards are false transient outputs of combinational circuits that occur when the outputs are changing. That is, a dynamic hazard occurs when, in response to some input change, an output that is designed to change only once changes three or more times. There are two derivatives of dynamic hazards, depending on if the output change is a low-to-high transition, or a high-to-low transition. Examples of low-to-high dynamic hazards and high-to-low dynamic hazards are presented in Figure 2.3 and Figure 2.4 respectively.

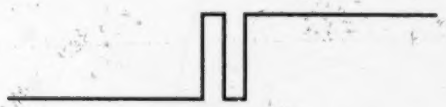

Figure 2.3. Low-to-high dynamic hazard [Roth79].

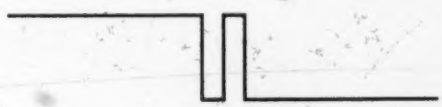

Figure 2.4. High-to-low dynamic hazard [Roth79].

2.1.3. Essential Hazards

Unlike race or combinational hazards, essential hazards appear only in AFSMs. These hazards occur when an input variable that causes a change of state reaches some parts of the circuit before it reaches other parts.

The presence of an essential hazard depends only on the structure of the original flow table. Consider the flow table shown in Figure 2.5.

	X	
	0	1
S1	S1	S2
S2	S3	S2
S3	S3	S3

Figure 2.5. Example flow table with essential hazard

Assuming the circuit is initially stable in state S1 with x = 0, when x changes to 1, the machine should change to state S2 and become stable. Because of delays, a portion of the circuit may receive the change in the state variable that indicates the machine is in state S2 before this portion receives the change of x from 0 to 1. Therefore, the portion of the circuit in which x is delayed behaves as if it is in state S2 with x = 0. This may cause the machine to incorrectly go into state S3.

This malfunction stems from the fact that the feedback signal reached a part of the circuit before the changing input signal. However, this signal timing does not cause a malfunction in every case. Consider the flow table shown in Figure 2.6.

	X	
	0	1
S1	S1	S2
S2	S1	S2

Figure 2.6. Example flow table without essential hazard

Assume that the circuit is stable in state S1. If x changes from 0 to 1, and the machine circuitry causes one state variable to change, the next state will be state S2. If a part of the circuitry sees this state change to S2, and has not yet seen the low to high transition on x, it will operate as if the final state is S1, until x propagates to it, at which time it will change to state S2. In this case, even though the feedback variable arrives before the input variable, there is no circuit malfunction.

This phenomena was first analysed by Unger [Unge59], and later by McCluskey [Mccl65], and both came to the conclusion that feedback-input variable races that cause circuit malfunction can be identified by inspection of the flow table. This leads to the following definition of an essential hazard.

> *Definition 2.1 [Unge59]* A total state S and an input variable x represent an *essential hazard* for a flow table T if and only if, when the table is initially in state S, the state reached after one change in x is different from the state reached after three changes in x.

2.1.4. Elimination of Hazards

AFSMs will not operate correctly until all of the hazard types described in the previous sections are solved. Solving these hazards requires additional design steps which are described in the following sections.

2.1.4.1. Solving Critical Race Hazards

Race hazards between state variables and input variables can be solved by:

1. Making sure there are no races between the variables or,
2. Making sure that any races between the variables are not critical

Ensuring that there are no races between *input* variables is accomplished by restricting the input variables to follow the Fundamental Mode Assumption [Mccl65]. Here, it is assumed that the circuit is allowed to fully stabilize after an input changes before another input is allowed to change.

It is possible to design working AFSMs that do not need to follow the fundamental mode assumption. In these circuits, the races between input variables are carefully designed to be non-critical. Cir-

cuits of this type are described in [Hack71] and [Sing68].

Ensuring that there are no races between *state variables* can be done by using the Multiple Transition Time (MTT) state assignment [Roth79]. Ensuring that any races between state variables are non-critical can be done by using the Single Transition Time (STT) state assignment [Lewi71]. MTT assignments avoid races altogether by forcing the state vectors of adjacent states to differ only by one bit. In contrast, the state vectors in STT assignments are carefully chosen so that any races in the state variables will not change the final state, and hence, the operation of the circuit. Several methods for generating STT assignments are discussed in [Kuhl78] and [Nany79].

2.1.4.2. Removing Combinational Hazards

Early work by Unger [Unge59] and McCluskey[Mccl65] determined that static and dynamic hazards could be eliminated from any sum-of-products combinational circuit by adding redundant minterms to the equation. McCluskey's proof of this can be found in Chapter 7 of [Mccl65].

Factoring of combinational circuits after static hazards are removed must be done with care. In page 193 of [Lewi74], Lewin shows an example of a circuit in which the combinational hazards are removed using McCluskey's method, but exhibits a dynamic hazard after the circuit has been factored. However, in [Roth79], Roth shows that any sum of products expression in which combinational hazards have been removed can be factored without introducing new combinational hazards as long as each variable 'x' is treated independently of its complement during factorization. Close examination of Lewin's factorization techniques shows that variables and their complements were not treated independently, which explains the presence of the dynamic hazard in his example.

2.1.4.3. Removing the Effects of Essential Hazards

Essential hazards cannot be removed from a design without restructuring the flow table. In addition, it may be impossible to remove all essential hazards from a flow table and keep the circuit behaviour desired by the user.

However, as previously outlined, the effects of essential hazards can be hidden by making sure that the inputs propagate to all parts of the circuit before the changing state variable does. This is typically done by placing delays in the feedback paths of all appropriate state variables. Delaying the state variables has the disadvantage of slowing circuit and removing some of the speed advantages that asynchronous circuits have over their synchronous counterparts. However, as first reported by Armstrong [Arms68], and later by Hackbart [Hack71], the effects of essential hazards can be removed by adjusting the path delays of particular input and feedback variables, using the gate delays to ensure that the inputs propagate to all circuit parts before the feedback does. This method has the advantage of not requiring delay elements, which can result in faster circuits.

2.1.5. Classical AFSM Theory Used in ADACC

As stated in Chapter 1, the main motivation behind ADACC is to see if rule-based timing optimization can be used in conjunction with classical AFSM theory to automatically generate asynchronous circuits in which correct operation is ensured by having timing optimization try to meet timing constraints. Therefore, ADACC uses most of the classical AFSM theory presented in this section, with the exception of some of the hazard removal methods. The hazard removal methods that are used in ADACC are outlined in the following paragraphs, along with explanations of why they are used over other methods.

Critical race hazards are removed in ADACC by eliminating all races in the input and state variables. This is accomplished by forcing the input variables to follow the fundamental mode assumption, and by using MTT state variable assignments. Elimination of all races was chosen over elimination of only critical races because of its simplicity of implementation.

The method used to eliminate combinational hazards involves inserting redundant product terms in all Boolean equations in the circuit.

Essential hazards are removed by adjusting circuit path delays rather than inserting delay elements into the feedback paths. As stated previously, this is done by using rule-based timing optimization to adjust circuit delays to meet particular timing restrictions.

2.2. Speed-Independent Circuits

Asynchronous Finite State Machines are not the only class of asynchronous circuits. Another important class of asynchronous circuits is called *Speed-independent circuits*. This class of circuits was first reported by Muller in [Mull65]. Other references of this class of circuits include Unger, [Unge69], and Seitz in Chapter 7 of [Mead80].

Correct operation of these circuits is not dependent on individual gate delays, and therefore these circuits are called *delay-insensitive*. Delay insensitivity is accomplished by making each block of the circuit generate a completion signal. These signals become true when a circuit block is done processing and its output is valid. Completion signals are then used to help generate handshake signals that are used to control the flow of information between circuit blocks. Typically, both the input and output port of a block are handshaked to ensure that information is passed to the next block when it is requested, and that outputs stay valid until the next block has used them.

Many of these circuits depend on a special logic gate, called a "c-element" [Mull65, Mead80, Meng89] to generate the handshake signals. A c-element is a two-state asynchronous circuit with two or more inputs and one output. Its output becomes 0 when all of its inputs are 0, and becomes 1 when all of its inputs are 1, and otherwise, the output stays in its previous condition. This circuit element is typically used to determine when all the required handshake events occur by connecting handshake signals to the inputs of the element, and then monitoring the output of the element. Because of this application, c-elements are also known as "last-of" circuits [Mead80], because they become true when the "last of" a set of handshake signals are received.

The handshake signals between circuit blocks are usually implemented as two signals, a *request* signal, and an *acknowledge* signal[Mead80]. The request signal is used both to request the transfer of information into a circuit block, and to signal when that block is done with the information. The acknowledge signal is used to inform a block that requested information is now available (valid). This controls the input timing and the output timing of a block. A good example of a system that uses request and acknowledge handshake is the VME bus handshake mechanism [Moto85]. Here, the request signal is implemented with the Address Strobe (AS) signal, and the acknowledge signal is implemented with the Data Transfer Acknowledge (DTACK) signal.

In order for speed-independent circuits to operate correctly, the input signals to the system must be constrained in the same manner as internal signals of the circuit. That is, inputs must only change in response to request signals from the circuit. This dictates that the surrounding circuitry must operate in a speed-independent manner, or some buffer circuitry is required at the inputs of the circuit to ensure that the inputs change only in response to input requests.

2.2.1. Synthesis of Speed-Independent Circuits

Recently, there has been much research done on the synthesis of speed-independent or self-timed circuits. Some of the earliest work in this field was done by Chu in [Chu86]. In this paper, Chu showed that self-timed circuits can be synthesized from a high-level description based on petri-nets. Petri-nets are an ideal mechanism for describing self-timed circuits because they can describe a series of events that must happen in a particular order. Chu described transitions of the input and output signals as events in petri-nets, and then used these nets to create the required completion and acknowledge signals. The Boolean expressions of these completion and output signals could be written down directly from the petri-net description. However, various restrictions must be applied to these petri-nets and the events that they describe in order for this synthesis system to work. The interested reader can see [Chu86] for more information.

This approach has been expanded upon by Borriello [Borr87, Borr88], and recently by Meng et al [Meng89]. Borriello uses 'interface event graphs', which are extended petri-nets that include timing constraints as well as event ordering information. Meng uses pure petri-nets, but has designed a working system that synthesizes asynchronous interconnection circuits for use in connecting synchronous VLSI subsystems that use different clocks.

2.2.2. Disadvantages of Speed-Independent Circuits

Although speed-independent circuit theory offers the advantage that circuit timing does not need to be analysed (or even worried about) to guarantee correct operation, they have other disadvantages.

First, speed-independent circuit theory is based on the assumption that the c-element is an ideal device that has zero propagation delay and is not susceptible to common asynchronous hazards such as critical races between inputs. In actual applications, c-elements are designed to be as fast as possible, and the rest of the circuit is assumed to be slower than the c-element[Meng89]. We believe that this assumption is dangerous design practice. In ADACC, the opposite approach is taken. Nothing is assumed to be ideal and the timing of the entire circuit is carefully controlled. This eliminates any assumptions regarding the timing of fundamental circuit blocks.

Second, the completion signals of circuit blocks are used to generate the handshake signals which are responsible for correct timing. The problem is that generation of completion signals of non-trivial circuit blocks (e.g. a 32 bit generate propagate adder) is very difficult. In [Meng89] it was shown that a DCVSL combinational circuit structure can generate its own completion signal as a side effect. However, this structure (which is based on a n-mos pulldown tree) cannot be used to implement complex combinational blocks, and requires differential inputs and outputs that can take up to 40% more routing area [Meng89].

Because of these disadvantages, we feel that although the use of self-timed circuits is a promising approach, more work is required in this area. Therefore, this research is focused on the automaton of

the design of classical AFSMs.

2.3. Rule-Based Logic Synthesis

Logic synthesis systems are a branch of knowledge-based systems that are intended to automatically generate logic circuits from high-level behavioural descriptions. This section describes the general theory behind rule-based systems, as well as two examples of successful rule-based logic synthesis systems, LSS and SOCRATES.

2.3.1. General Rule-Based Systems [Rich83]

Rule-based systems are comprised of three sections:

1. Data. This section contains design information. That is, it contains all the information about the current design that the system is in the process of modifying so that it meets the user's requirements.
2. Knowledge Base. This section contains knowledge that the system is going to use to help meet the requirements. This knowledge is usually in the form of modus ponens rules that are individually applied to the design. These rules are pattern matching rules: a rule can be applied if the left hand side of the rule matches some part of the design. When a rule is applied, the part of the design that was matched with the left hand side is deleted and replaced with a copy of the information on the right hand side.
3. Control. The control section decides which rules should be applied to which parts of the design, and in what order. Naturally, such a control mechanism requires at least one method of measuring the quality of the design before and after a rule is applied in order to help rank desirable rules. Heuristics are often used to help reduce the size of the search space by pruning sequences of rules that are unlikely to produce a good solution. The control section also determines when the user's requirements are satisfied, or if these requirements cannot be met.

The state space is the set of all possible variations of the Data section. Searching the state space is done by appling rules to modify the Data section. Heuristics in the Control section are used to pick the rules to apply, which is equivalent to guiding the search of the state space.

In rule-based logic synthesis systems, the Data portion is a circuit description, usually in the form of a netlist. The rules in the Knowledge Base contain sub-circuits on both sides. These subcircuits are functionally equivalent, but the sub-circuit on the right hand side is in some way preferable to the sub-circuit on the left hand side.

The following sections describe some of the research done with rule-based logic synthesis.

2.3.2. LSS

LSS [Darr81], [Darr84] is one of the first rule-based logic synthesis systems. It was originally developed as a research project at IBM to help implement the control portion of synchronous machines and, in general, any random logic. It was hoped that a high-level functional description could be used as a starting point, and that the system would produce low-level multilevel logic circuits that met timing, area, and technology constraints.

The system operated by first transforming the high-level specification to an initial circuit description. Then, a series of local rules (called transformations in LSS) were applied, each transformation changing the circuit so that it better met the specified constraints.

Unlike later systems (e.g., SOCRATES [Geus85]), the initial version of LSS was built as an interactive system. The designer had total control over where and when a particular rule was to be applied, rather than using an automated search procedure to choose a transformation. That is, the user acted as the Control portion of the system. There were numerous evaluation programs available to the user that determine how a particular transformation will affect the circuit, but the user still made the final decision on which transformation to apply, and where in the circuit it was to be applied. The initial version of LSS reported in [Darr81] was found to produce good results, using 0% to 15% more

gates than similar hand-designed systems.

In [Darr84], Darranger describes the changes made to LSS in order to make it into an acceptable production tool for an IBM CPU project. Here, LSS was used to create approximately 90 IBM masterslice chips.

This version of LSS was made non-interactive by automating rule selection. This was done by creating a library of "scenarios", which were simply lists of rules that seemed to work well with particular design types. However, the user chose the scenario to be used in synthesis and thus still selected the rules to be applied. Although this was a fast way to make LSS automatic, intelligence in rule selection was degraded, and the resulting circuit quality undoubtedly suffered. In order to produce acceptable results, LSS was extensively modified. However, it retained the basic notion of using local transformations to meet design restrictions, which showed that the initial idea had merit.

2.3.3. SOCRATES

The SOCRATES [Geus85] system was another rule-based logic synthesis system designed for random combinational logic. The goal of this system was to produce minimum multilevel circuits that met the user's timing restrictions, and various technology restrictions such as fan-out and input load.

Like LSS, SOCRATES used local transformations that were in the form of modus ponens rules. However, the main difference between SOCRATES and LSS is that SOCRATES used an extensive heuristic search method to implement the control section. This control section found the rules or sequences of rules that were applicable to the circuit, and of these determined which was more desirable. The rule or rules that contributed to the largest improvement of the circuit were then applied.

The control section searched for sequences of rules so that it was possible for some uphill rules (i.e. rules that degraded the circuit) could be included in a sequence of rules that as a whole improved the circuit. This was done so that the system would not get stuck in a local minimum solution.

These rule sequences were determined by exploring a breadth and depth limited search tree. In addition, the breadth and depth limits of this tree were updated concurrently by a series of "meta-rules" that determined if the search would benefit from expanding the search space.

Because of this intelligent control mechanism, and an extensive knowledge base, SOCRATES performed very well, managing to produce some circuits that were superior to hand-designed circuits. This performance increase over LSS can be directly attributed to the greater intelligence of the rule selection mechanism.

2.3.4. Logic Synthesis Theory Used in ADACC

The main goal of logic synthesis in ADACC is to eliminate most if not all essential hazard effects by adjusting circuitry delays. However, as shown in [Lewi74], it is possible to introduce combinational hazards if particular factoring forms are used in synthesis. Therefore, it is desirable to have complete control over the types of transformations performed by synthesis to avoid the addition of combinational hazards. Because of this consideration, rule-based synthesis methods are an ideal choice for implementation of ADACC's logic synthesis system.

However, it is beyond the scope of this thesis to implement a commercial quality logic synthesizer. Therefore, ADACC uses a limited synthesis system, which is called a rule-based timing optimizer. This optimizer has all the elements of a logic synthesis system, but it is limited in that it at times must resort to the use of rules that modify gate delays rather than change the circuit structure, which is not logic synthesis in the strict sense. These limitations are further described in Chapters 3 and 4 of this thesis, and the additional benefits of incorporating a more sophisticated logic synthesis system into ADACC are discussed in Chapter 7.

3. High-Level Overview of ADACC

3.1. Introduction

In this chapter, a brief description of each module of ADACC is presented, including what is accomplished in each particular module, and most importantly, the theory behind how it is accomplished. The implementation details of ADACC are presented in Chapter 4.

The main portions of ADACC are:

1. User State Machine Input - reads textual description of the initial user state machine.
2. State Assignment - generates race-free state assignments.
3. Equation Generator - generates Boolean equations for output and state variables.
4. Combinational Hazard Remover - removes combinational hazards from the generated equations.
5. Essential Hazard Detector - detects and records essential hazards.
6. Timing Restriction Generator - determines path restrictions required to meet user timing constraints and to remove effects of essential hazards.
7. Rule-Based Timing Optimizer - creates final circuitry and adjusts path delays to hide essential hazards, and to meet most of the user timing restrictions.

These portions are further described in the following sections.

3.2. User State Machine Input

Before ADACC can perform any synthesis, the state machine specification must be entered into the system by the user. The input format used is an ASCII representation of the state machine read through a UNIX standard input. The user specifies the names of the input and output variables, the states in the FSM, the behaviour of the outputs in each state, and the transitions between states. In addition to specifying the next state equation, the user can constrain the maximum time in nanoseconds that any transition is allowed to take, as well as a global default transition time that represents the maximum time in nanoseconds that *every* transition in the state machine is allowed to take. The syntax used to parse the user's input is described in Appendix A and an example input file is shown in Appendix E.

3.3. State Assignment

ADACC uses Multiple Transition Time (MTT) state assignments as these assignments eliminate the chance of critical races between state variables by not allowing any races. This is accomplished by allowing only one state variable to change during a state transition.

With MTT assignments, the state assignment problem is reduced to assigning every state a unique bit vector such that bit vectors of states that are connected with a transition differ by only one bit. Unfortunately, MTT assignments usually require transition or shared states to ensure that the final assignment is race-free. MTT assignments have long been criticized for the performance penalties these states add. However, in ADACC, the user specifies timing constraints on transitions that must be fast. These timing constraints are used in the state assignment to avoid placing transition states into time restricted transitions. Therefore, any required transition states are placed to avoid penalizing the user's specified speed requirements.

State assignment is performed in two steps:

1. Use simulated annealing to create a good initial assignment.
2. Clean up the assignment produced by annealing to guarantee that the final assignment is race-free.

Annealing creates an assignment that is as free of races as possible without including transition states, and avoids violating the user's timing restrictions. This is accomplished by weighting the annealing cost function to avoid assigning non-race-free assignments to transitions that have a small user-specified transition time. The cleanup procedure then adds transition or shared states to any transitions that were not assigned race-free assignments by annealing.

3.4. Equation Generation

After state assignment has been completed, equations for the output and next state variables are created. The output equations are generated from the state vectors and the output behavioural equations in each state. The next state equations are generated from the state vectors and next state equations

associated with state transitions. The following sub-sections further describe the creation of these equations.

3.4.1. Generation of Output Equations

The equations that describe the output functions are derived using the following formula:

$$output_m = (SV1 * O_m1) + (SV2 * O_m2) + ... + (SVn * O_mn)$$

Here, 'SV1', 'SV2', and 'SVn' are Boolean equations that represent the assignments of states 1, 2 and 'n', and 'O_m1', 'O_m2', and 'O_mn' are the Boolean equations of the output 'm' in those states. These Boolean output functions can be constant (for Moore-type FSMs), or can be a function of one or more input variables (for Mealy-type FSMs). For example, assume that we start with the two state machine shown in Figure 3.1.

Figure 3.1. Example state machine

With this state machine, the Boolean equation for the output 'out1' would be:

```
out1  = (Q1 * (in1 + in2)) + (!Q1 * (!in1 * in3))
      = (Q1 * in1) + (Q1 * in2) + (!Q1 * !in1 * in3)
```

3.4.2. Generation of Next State Equations

The generation of state variable equations is considerably more complex than the generation of output equations. ADACC uses the Zissos algorithm [Ziss79] for creating the state variable equations. This algorithm can be expressed using the following equation:

Qn+ = (sum of turn-on sets of Qn) + Qn * !(sum of turn-off sets of Qn)

Here, 'Qn+' represents the 'next state' of the state variable that is to be calculated, and 'Qn' represents the 'previous state' of that variable. The expressions 'turn-on set' and 'turn-off set' are defined below:

Definition 3.1 [Ziss79] The turn-on set of a state variable is a set of Boolean variables, which when equal to a logic 1, cause the state variable to turn on.

Definition 3.2 [Ziss79] The turn-off set of a state variable is a set of Boolean variables, which when equal to a logic 1, cause the state variable to turn off.

An example of how the turn-on and turn-off sets of a state variable are used to calculate the next state equations follows. Assume that we start with the FSM shown in Figure 3.2.

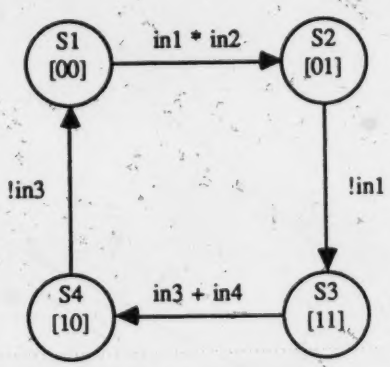

Figure 3.2. Example state machine

Here, the Boolean values in square brackets are the assignments of the state variables 'Q1' and 'Q2'. The state variable 'Q1' turns on during the state transition from state S2 to state S3. This transition occurs when the input 'in1' is equal to logic 0, and when 'Q2' is equal to logic 1; therefore, the turn-on set of 'Q1' is (!in1, Q2). 'Q1' turns off during the transition from S4 to S1, which occurs when the input 'in3' is equal to logic 0, and 'Q2' is equal to logic 0. Therefore, the turn-off set of 'Q1' is (!in3, !Q2). Similarly, the turn-on set of 'Q2' is (in1, in2, !Q1), and the turn-off set of 'Q2' is ((in3+in4), Q1). This leads to the following next state equations of 'Q1' and 'Q2':

```
Q1+  = (!in1*Q2) + Q1*!(!in3 * !Q2)
     = !in1*Q2 + Q1*in3 + Q1*Q2

Q2+  = (in1*in2*!Q1) + Q2*!((in3+in4) * Q1)
     = in1*in2*!Q1 + Q2*!in3*!in4 + Q2*!in3*!Q1 + Q2*!Q1*!in4 + Q2*!Q1
```

After all equations are generated, they are reduced to minimum sum of products expressions with the ESPRESSO [Bray84] logic minimizer.

3.5. Combinational Hazard Remover

After reduction, the combinational hazards associated with the resulting equations must be removed. As shown by McCluskey [Mccl65], a Boolean equation will be free of combinational hazards if all of the adjacent minterms in that equation are 'covered' by at least one additional product term. For example, the following equation contains a combinational hazard:

```
op = b*!c*d + a*c*d
```

Here the two terms (b*!c*d) and (a*c*d) are adjacent because the term (b*!c*d) includes the input vector (abcd = 1101), which is adjacent to the input vector (abcd = 1111) included by the term (a*c*d). The equation can be made free of combinational hazards by including the product term (a*b*d), which includes both adjacent input vectors and thus covers the original minterms. Therefore, to remove combinational hazards, we need to identify all adjacent minterms, and then add additional products that cover these terms.

3.6. Essential Hazard Detector

As described in Chapter 2, essential hazards in an AFSM can be detected by inspection of the flow table: if the final state after an input variable is toggled once is different than the final state after that input is toggled three times, then an essential hazard exists. Therefore, the essential hazard detector must go through every stable total state in the FSM, and toggle every input three times and check the

final states reached after one and after three toggles. The information needed to identify each essential hazard is as follows:

1. The input variable that, when toggled, causes the essential hazard.
2. The state variable that changes during the transition caused by the first toggle of the input variable.
3. The total state of the FSM before the input variable is toggled.

This information is later used to create timing restrictions which are used in rule-based optimization to help eliminate essential hazard effects.

3.7. Timing Restriction Generator

This section describes the timing restrictions that must be placed on the circuit in order for it to meet most of the user's timing constraints and the constraints required to avoid essential hazard effects.

It should be noted that ADACC does not meet all of the user's timing constraints. This is because of transition states inserted during state assignment. However, this problem is identified and a possible solution is presented in Chapter 7.

All of the timing constraints can be described using paths in the synthesized circuitry. A circuit path is a set of gates that a change in an input signal must propagate through on its way to an output signal. Adding the delays of every gate in the path gives the delay of that path. Of course, there could be several different paths that a signal could take, depending on the state of the circuit during the signal propagation. Therefore, some method is needed to pick the proper path to use in timing optimization. The method used in ADACC is simply to use either the longest or the shortest delay path, depending on the timing constraint that is being evaluated. For example, if the timing constraint specified that the path delay between a specific input and output be longer than a particular value, the shortest delay path connecting the input and output would be found, timed, and compared to this value.

The following sections describe how the timing restrictions are specified as a series of paths in the circuit.

3.7.1. Default Transition Time Restrictions

The default transition time is a user timing constraint that represents the maximum time that any state transitions is allowed to take. Another way of looking at this is to think of this time as the time it takes the circuit to reach steady state after an input change. Therefore, the default transition time is the minimum time between input changes that can be tolerated and not violate the fundamental mode assumption [Roth79].

The default transition time should be equal to the maximum time it takes for changes in any input to propagate to every state variable, plus the time it takes for changes in any state variable to propagate to every other state variable. This can be explained further with the help of the following definitions:

Definition 3.3 Primary Path. A primary path is the path from the input variable that is causing a state change to the state variable that is to change in the state transition.

Definition 3.4 Feedback Path. A feedback path is a path from the state variable that is changing due to a state transition to one of the other state variables.

Figure 3.3 shows the primary path from the input 'Ii' to the next state variable 'Qn', and it shows the feedback path from the next state variable 'Qn' to the next state variable 'Qm'. Note that state variables 'Qn' and 'Qm' are not equal.

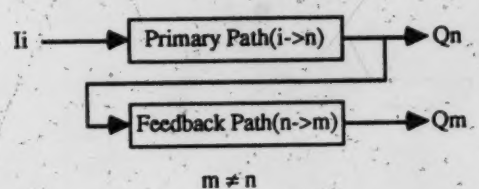

Figure 3.3. Primary and feedback paths

With the above definitions, the timing restrictions required for the circuit to meet the default transition time can be written as:

$$\lceil \text{primary_path}_{(i \rightarrow n)} \rceil + \lceil \text{feedback_path}_{(n \rightarrow m)} \rceil \leq \text{Default_tt}$$

for all i, n and m such that $m \neq n$.

Or in words, the transition times of all state transitions will not be longer than the user-specified default transition time if the sum of the slowest primary path delay and the slowest feedback path delay is less than the user specified time.

3.7.2. Transition Time Restrictions

The transition time restriction is similar to the default transition time restriction with the exception that only one state transition is involved. Therefore only the primary path between the input variable that initiates the transition and the state variable that changes in the transition is needed to represent the timing restriction. This can be written as:

$$\lceil \text{primary_path}_{(i \rightarrow n)} \rceil + \lceil \text{feedback_path}_{(n \rightarrow m)} \rceil \leq \text{transition_time}$$

for all m such that $m \neq n$, where n is the state variable that changes during the transition, and i is the input that initiates the transition.

Here, the primary path is the longest path between the input that initiates the state transition and the state variable that changes during the transition. The feedback path is the longest path of all the paths between the changing state variable and all the other state variables.

Note that it is assumed that the transition time is less than the default transition time. Transition times larger than the default transition time can be specified, but they will have no effect on speed performance as all transitions will be made to be at least as fast as the default transition time.

3.7.3. Essential Hazard Timing Restrictions

The timing restrictions required to avoid the effects of essential hazards involve making sure that one path delay is faster than another. An additional definition is required here to help specify the paths involved:

> *Definition 3.5* Secondary Path. A secondary path is a path from an input variable that is causing a change of state to any of the state variables that are NOT changing due to the state transition.

Therefore, if there are M state variables, there are M-1 secondary paths from the changing input to all the state variables that are not changing in the state transition. The illustration presented in Figure 3.4 describes this further.

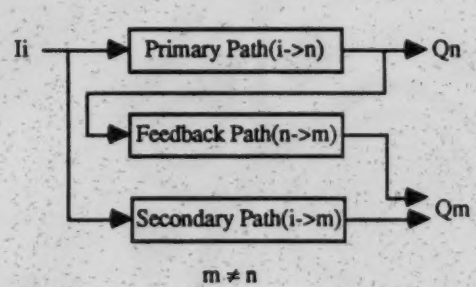

Figure 3.4. Secondary path

If an essential hazard occurs when toggling the input variable 'Ii', then to remove the effects of this hazard we must guarantee that:

$$\lfloor \text{primary_path}_{(i \to n)} \rfloor + \lfloor \text{feedback_path}_{(n \to m)} \rfloor > \lceil \text{secondary_path}_{(i \to m)} \rceil$$

for all m such that $m \neq n$.

That is, we must make sure that the sum of the shortest path between the input variable Ii and the changing state variable Qn, and the shortest path between Qn and each other state variable Qm ($m \neq n$) must be longer than the longest path from Ii to Qm.

3.8. Rule-Based Timing Optimization

In this section, the possibility of using rule-based optimization to meet user timing constraints and to remove essential hazard effects is discussed.

3.8.1. Use of Rule-Based Optimization to Hide Essential Hazard Effects

As previously noted, Armstrong[Arms68] triggered a series of papers aimed at removing the effects of essential hazards without adding delay elements in the feedback path. In this paper, Armstrong showed how "special factoring" is used to "...ensure that the x (input) variable change is seen by the first level gates before any y (state) variable change is seen" [Arms68].

Armstrong's solution showed that by adjusting path delays, it is possible to guarantee that changing state variables will be delayed enough so that malfunctions due to essential hazards will not occur. Rule-based optimization, among other features, has the ability to modify circuit path delays to follow any timing restriction. It stands to reason that if the required path delays can be specified as a set of timing restrictions, rule-based optimization can be used to modify the circuit so that it meets these timing restrictions. After this is done, the resulting circuitry should not exhibit any malfunctions due to essential hazards.

3.8.2. Requirements of Rule-Based Optimization System

ADACC's timing optimizer must use the timing restrictions described in section 3.7 to modify the circuit to remove essential hazard effects and to meet most of the original user timing constraints. Meeting these restrictions is the main function of the rule-based optimization system.

Rule-based optimization should be the last step in synthesizing the AFSM circuitry, as it involves adjusting circuit paths and any additional processing could upset the timing of these paths. Therefore, the optimization step must occur after combinational hazards are removed. This creates an extra requirement in that the optimizaton must not add any combinational hazards to the circuit. In summary,

rule-based optimization has the following requirements:

1. must remove the effects of most (if not all) essential hazards by following given timing constraints.
2. must adjust circuit timing so that most (if not all) of the original user timing constraints are followed.
3. must not introduce any combinational hazards.

3.8.3. Selecting The Rule Base

On page 193 of [Lewi74], Lewin shows how careless factoring of a combinational circuit can introduce dynamic hazards. In [Roth79] Roth states that adding combinational hazards while factoring can be avoided by using factoring rules that assume each variable 'x' is treated independently of its complement !x. For example, the rules

```
x * !x = 0
x + !x = 1
(x * y) + (!x * z) = (x + z) * (!x + y)
```

do not treat x and !x independently, and hence can introduce hazards if used to optimize an AFSM. Therefore, the optimization rules used must not assume that a variable and its complement are dependent.

This restriction has been used to evaluate Boolean algebra rules by [Roth79], [Ziss79], and [Mccl86] to derive a set of Boolean rules that are suitable for use in the optimizer. This rule set is shown in Figure 3.5.

	Rule:
1	x+0→x
2	x*0→0
3	x+1→1
4	x*1→x
5	x*x→x
6	x+x→x
7	(x*y)*z→x*(y*z)
8	(x+y)+z→x+(y+z)
9	x+(x*y)→x
10	x*(x+y)→x
11	x*(y+z)→x*y + x*z
12	x+(y*z)→(x+y)*(x+z)
13	!(x+y)→!x*!y
14	!(x*y)→(!x+!y)

Figure 3.5. Boolean algebra based topographical optimization rules

None of these rules assume that x and !x are the same variable, and thus they follow Roth's conditions. It is easy to show that rules 1 - 12 do not violate Roth's restriction, since none of them include both a variable and its negation. Rules 13 and 14 are DeMorgan's laws, which according to Roth, do not introduce hazards when applied.

The above rules are called topographical rules, because they modify the circuit's speed performance by changing the circuit structure. That is, these rules remove and add gates along particular paths to change the overall circuit timing without destroying the circuit's function.

The above rule set is derived from Boolean algebra identities, and thus assumes that the technology used to implement all of the gates is the same. However, additional timing rules can be written if the technology used allows the user to vary these gate properties. For example, a rule can be written to allow the substitution of a two input TTL AND gate with a two input high-speed CMOS AND gate. These gates perform the same logical function, but have different timing characteristics. Such rules do not violate Roth's restrictions, and therefore rules such as the ones shown in Figure 3.6 can be used in timing optimization as well.

	Rule:
1	TTL_AND→CMOS_AND
2	CMOS_AND→High_speed_CMOS_AND
3	High_speed_CMOS_AND→GaAS_AND

Figure 3.6. Gate delay optimization rules

These rules are called gate delay rules, because they adjust the circuit timing by simply changing the gates along particular paths, which modifies the delays of these paths. These rules have the disadvantage that they combine several different gate types while topographical rules change the circuit structure using a fixed set of gate types. In addition, using gate delay rules to modify path delays is almost the same as adding delays in these paths. Therefore a system that only uses gate delay rules is not as interesting as a system that uses topographical rules. For these reasons, gate delay rules are not as desirable as topographical rules.

Although the use of gate delay rules should be avoided, both topographical and gate delay rules are used in timing optimization. Topographical rules are first used in timing optimization to remove most (if not all) timing violations. Then gate delay rules are used to clean up any timing problems that are left over. This method of using two rule bases was adopted for the following reasons:

1. ADACC's rule-based optimization system is limited in power, both in the control section (only downhill moves are used) and in the topographical rule base (only rules that do not add hazards can be used). In addition, topographical rules that affect more than one gate in a circuit may affect the timing of more than one path. Therefore a rule that is used to meet one timing constraint may end up breaking another. Because of these problems, ADACC may not be able to remove all timing errors using topographical rules alone.
2. It is desirable to synthesize circuits without timing violations because the resulting circuitry will operate correctly. Because topographical rules may not eliminate all timing errors, gate delay rules may be required to finish the timing optimization of a circuit.

It is important to show that topographical rules are useful in eliminating essential hazard effects. However, it is also important to have final circuits that have no timing restrictions so that they will be correct. Therefore if the need arises, ADACC falls back upon gate delay rules to guarantee correct circuit timing.

4. Implementation of ADACC

4.1. Overview

ADACC is a large program made up of just under 20,000 lines of C++ [Stro86] code. Because of the size of ADACC, a complete implementation description will not be presented in this chapter. Only the parts of ADACC that are considered important or novel are presented here.

A block diagram of the important components of ADACC is shown in Figure 4.1.1. The state machine graph object, the circuit graph object, and additional important objects are described in section 4.2. The implementation of state assignment is described in section 4.3, and section 4.4 describes the implementation of the combinational hazard remover. Section 4.5 describes the implementation of the essential hazard detector, and section 4.6 describes the implementation of the rule-based timing optimization system.

4.2. Important Objects used in ADACC

ADACC was implemented in C++ [Stro86] to try to take advantage of the use of objects. The purpose of this section is to describe the important objects used in ADACC.

Programs written in a object-oriented programming environment generally consist of a set of objects. These objects consist of a private data area, some private code to operate on this data, and a set of methods that are used to send messages to and from the object. Objects are usually instances of 'generic objects' or *classes*. Objects are defined by specifying their class, and are created by generating an *instance* of a class. For more information regarding object oriented programming environments, please see [Stro86].

The main object classes used in ADACC are the graph class and the data class. All other major classes are either children of these classes, or are created using one or more of these classes.

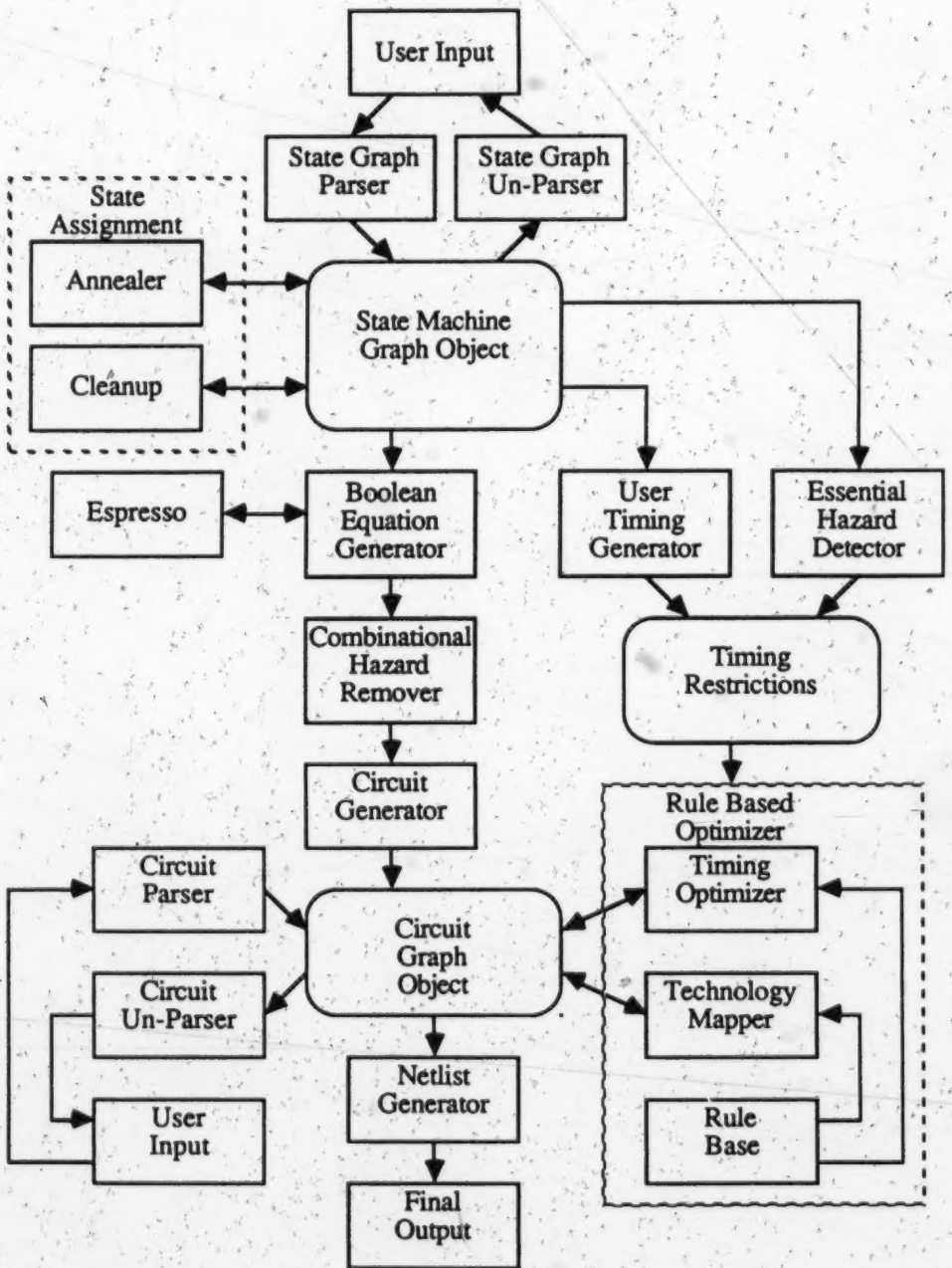

Figure 4.1.1. Block diagram of ADACC

4.2.1. Data Class

This is the parent class of all objects that hold data information. There are many children of this class. A few of the important ones described below are:

1. State Class. This class represents a state in a FSM.
2. Transition Class. This class represents a state transition in a FSM.
3. Equation Class. This class represents Boolean equations in a sum-of-products form. The data representation used is based on that used in ESPRESSO [Bray84]. This class includes methods for Boolean operations such as AND, NOT, and OR, as well as Boolean reduction.
4. Product Class. Used to represent a single product term in objects of the equation class.
5. Gate Class. Members of this class represent physical logic gates in the circuit graph object. The speeds of all gates used in ADACC can be found in Appendix B. There are five child classes of the gate class:
   - 5.1. Input port gate. Used to represent an input variable in the circuit.
   - 5.2. Output port gate. Used to represent an output variable or a state variable in the circuit.
   - 5.3. OR gate. Used to represent an OR gate in the circuit.
   - 5.4. AND gate. Used to represent an AND gate in the circuit.
   - 5.5. Inverter gate. Used to represent an inverter in the circuit.
6. Wire Class. Members of this class represent gate interconnection wires in a circuit.
7. Hazard Class. Members of this class contain information describing essential hazards.
8. Timing Restriction Class. Members of this class hold the timing restrictions imposed on the circuit by essential hazards and the user.

All of the children of the data class inherit a common set of basic methods which are listed in Appendix C.

4.2.2. Graph Class

The graph class implements a general directed graph structure. It is implemented with a set of node objects, a set of arc objects, and pointers that connect them. Arc objects were included rather than just connecting up the nodes with pointers because information must be associated with the arcs.

This class also contains a set of general methods that are used to add information, delete information, traverse the graph, and perform graph pattern matching.

Two types of graph traversal are permitted: sequential and topographical. Using sequential

traversal, all of the nodes and arcs in the graph can be traversed in the order in which they were added to the graph. This method is useful when all the nodes must be inspected in order. Using topographical traversal, the nodes and arcs can be traversed based on how they are connected to each other in the graph. The methods to perform traversal are further described in Appendix C.

The graph class has built-in parser and deparser systems. These allow the creation of any graph class from an ASCII representation of the graph and allow the graph to be saved in an ASCII file. Syntax descriptions of the state_graph parser and the circuit parser are presented in Appendix A.

4.2.3. Children of the Graph Class

The graph class has two children: the 'state_graph' class, used to hold a representation of the original FSM, and the 'circuit' class, used to hold a representation of the final circuit produced by synthesis.

Each node of a state_graph object is associated with a 'state' object, and each arc of the graph is associated with a 'transition' object. Each node of a circuit_graph object is associated with a 'gate' object, and each arc of the graph is associated with a 'wire' object.

Every circuit starts with a series of input port objects that represent the input variables. The outputs of these input ports are connected to the inputs of logic gates, the outputs of which are connected to more logic gates, until the output ports representing state or output variables are reached. As each gate in the circuit has only one output and one or more inputs, the circuits resemble triangles, or *cones of influence* [Darr84]. A typical circuit with its cone of influence is shown in Figure 4.2.1.

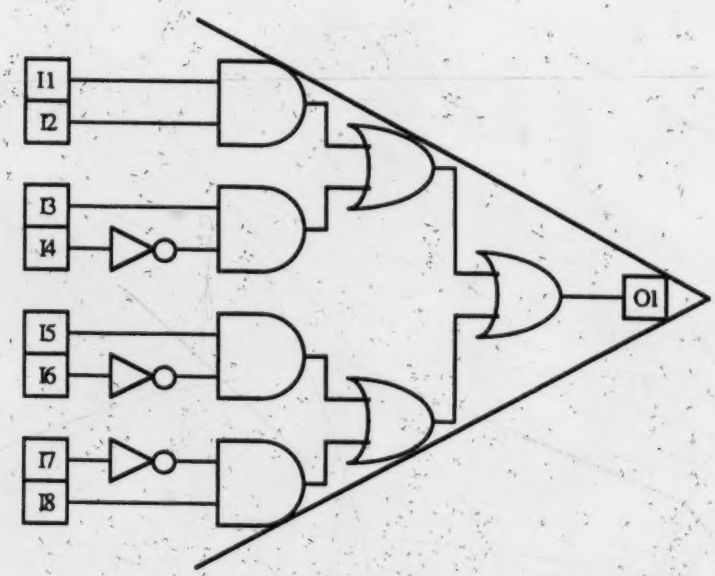

Fig 4.2.1. Typical cone of influence

In the circuit representation used here, every output or state variable has its own separate cone. That is, the logic used to produce an output is independent of the logic used to implement every other output; no sharing of gates is allowed. In some logic synthesis systems, such as SOCRATES[Geus85], the cones are allowed to overlap so that several logic blocks can share circuitry and therefore reduce the size of the overall circuit. Although this is desirable, it is beyond the scope of this research.

4.2.4. Rule Class

The rule class is used to represent the rules used by the rule-based timing optimizer. Therefore, this class must be able to represent all of the rule types presented in section 3.8.3 of chapter 3. These rules are represented as modus ponens substitution rules, with a Left-Hand-Side (LHS), and a Right-Hand-Side (RHS). Both the RHS and the LHS of each rule are objects of the circuit class; the RHS and the LHS are different implementations of the *same* logic function, which guarantees that the application of a rule will not change the basic function of the circuit. A simplified list of the rules used in optimi-

zation was presented in Chapter 3.

The gates in both sides of a rule require some additional information before they can be used in optimization. Each input and output port in the rule has an 'other_side' pointer. This pointer points to the equivalent input or output port on the opposite side of the rule. For example, if both sides of the rule have the inputs 'A', 'B', and 'C', there would be a pointer from the 'A' input port on the RHS to the 'A' input port on the LHS, and vice-versa. Similar pointers would exist for the 'B' and 'C' inputs, as well as any outputs in the circuit.

In addition to the 'other_side' pointer, each gate in the rule has a 'match' pointer as well. The match pointers in the LHS gates point to the corresponding gates in the circuit that the LHS is matched to. These gates would be deleted if the rule were to be applied. The match pointers in the RHS gates are set up after a rule is applied to point to the corresponding new gates that are added to the circuit. The RHS match pointers are used to undo rule applications.

These 'match' and 'other_side' pointers are used to keep track of the relation between both sides of the rule, and the relation between the rule and the circuit. Figure 4.2.2 shows the structure of a typical rule, and a circuit that it is matched to. The gates in the circuit drawn in bold are matched to the gates on the LHS of the rule.

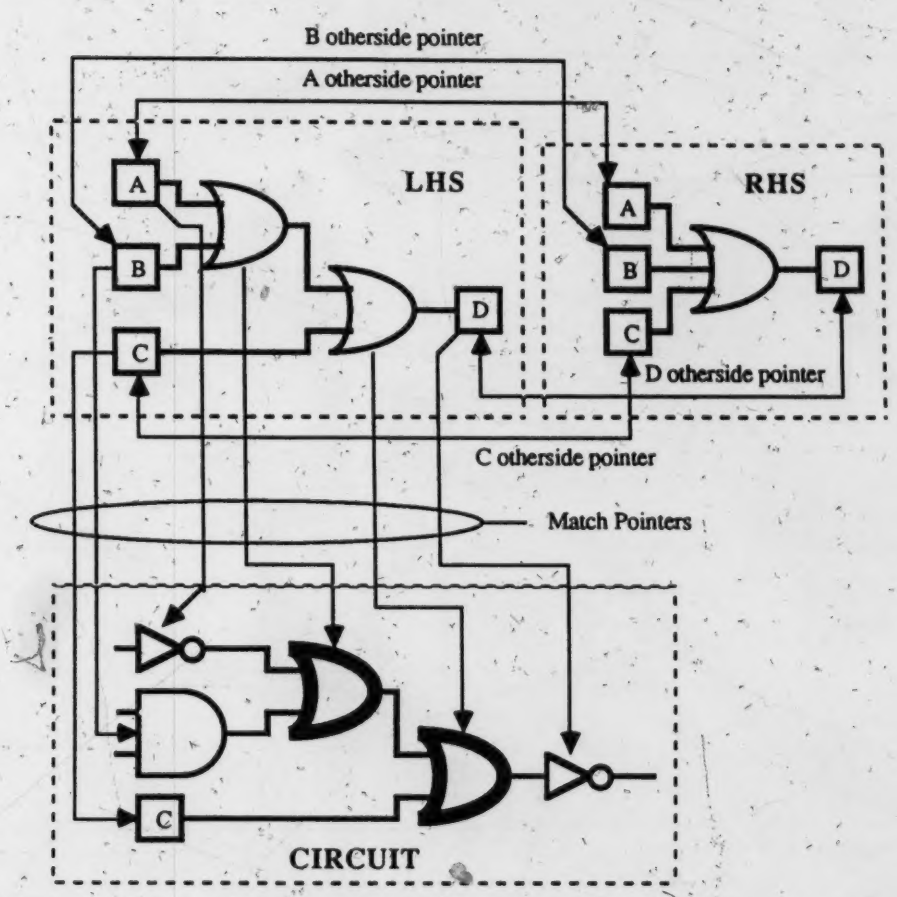

Figure 4.2.2. Pointer structure of a typical rule

The same rule can be matched to several different sets of gates in the circuit, assuming that there is a different copy of the rule for each set.

4.3. State Assignment Implementation

4.3.1. Overview

As stated in chapter 3, the assignment problem is twofold: assignments of adjacent states must be race-free and any required transition/shared states must be added in such a way that the user's timing

constraints are not violated.

The solution used in ADACC is to use simulated annealing [Kirk83] to get close to the best assignment that does not use transition states, and then apply a cleanup procedure to insert transition or shared states to make any non-adjacent transitions race-free.

Originally, ADACC's state assignment system also included a pre-processor that was run before annealing. This pre-processor, called the state splitter, was intended to split up states that had a large number of transitions associated with them. Unfortunately, the use of this pre-processor did not improve the quality of assignments produced by ADACC [May90-1] and therefore it was removed from the assignment system. For completeness, however, it is described in the following section.

4.3.2. State Splitting Pre-processor

State splitting was intended to split up any states that had too many transitions associated with them to be assignable with an assignment vector of fixed size. For example, a state with 4 inputs and 3 outputs (for a total of 7 connected states) will not be assignable if the size of the vector is 5.

State splitting was implemented with a simple search algorithm that found states that had too many transitions associated with them. Once these states were found, they were split up into two or more states, depending on the number of transitions associated with them.

State splitting is divided into two categories: top state splitting and bottom state splitting.

Top state splitting is required when the number of input transitions of a state is too large. When this condition occurs, additional states are added at 'the top' of the state. The input transitions are then distributed among these new states. The output transitions of the new states are attached to the original state. For example, assume that the state 'S0' with seven inputs shown in Figure 4.3.1a only has 4 state variables available. Top splitting causes additional states 'S1', 'S2', and 'S3' shown in Figure 4.3.1b to be added in order to reduce the number of arcs attached to each node to 4 or less. The next state equations of the new states are set up so that they are always true, while the next state equations of the

original transitions remain unchanged. The output equations of the states 'S1', 'S2', and 'S3' are set to be the same as the output equation of state 'S0'.

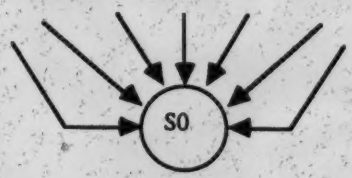

Figure 4.3.1a. State before top splitting

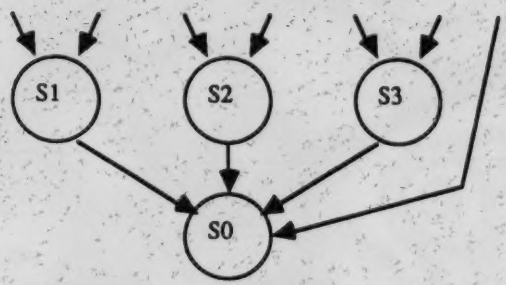

Figure 4.3.1b. State after top spliting

Figure 4.3.1. Top state splitting example

Bottom state splitting is required when the number of output transitions of a state is too large. When this condition occurs, additional states are added at 'the bottom' of the state in the same manner as the top split states. The output transitions of the original state are distributed among the new states. For example, assume that the state 'S0' shown in Figure 4.3.2a only has 4 state variables. The number of transitions leaving 'S0' must therefore be reduced to something less than 4. This causes the bottom splits 'S1', 'S2', and 'S3' shown in Figure 4.3.2b to be added.

The next state equations of 'S0' are modified to force correct operation of the circuit. For example, the next state equation leading to state 'S1' in Figure 4.3.2b is created by ORing together the next state equations leading to state 'Sa' and state 'Sb' in Figure 4.3.2a. This creates the next state equation (Ia + Ib), where 'Ia' is the next state equation leading to 'Sa' in Figure 4.3.2a, and Ib is the next state equation leading to 'Sb' in Figure 4.3.2a. The next state equation leading from state 'S1' to state 'Sa'

in Figure 4.3.2b is the same as the original next state equation leading from state 'S0' to state 'Sa' in Figure 4.3.2a. This is also true of the next state equation leading from state 'S1' to state 'Sb' in Figure 4.3.2b.

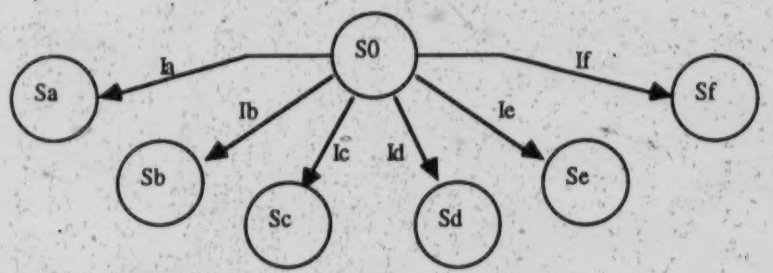

Figure 4.3.2a. States before bottom splitting

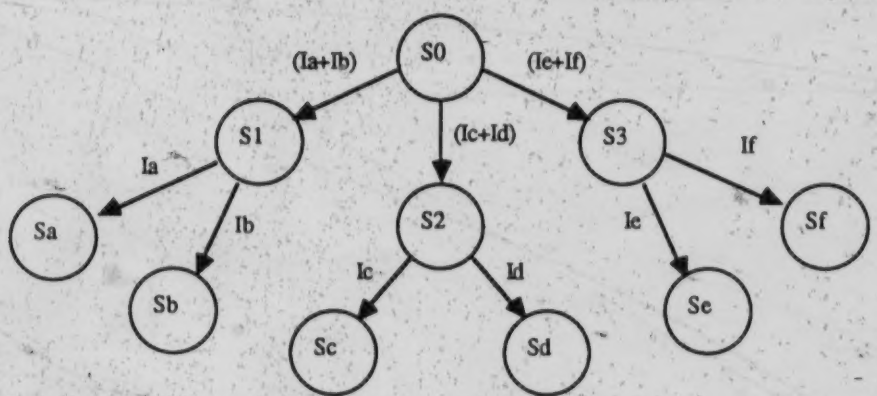

Figure 4.3.2b. States after bottom splitting

Figure 4.3.2. Bottom state splitting example

As stated previously, state splitting failed to improve the performance of ADACC's state assignment system. Therefore ADACC does not use state splitting or any other pre-processor in state assignment.

4.3.3. Initial Assignment with Simulating Annealing

Simulated annealing is a general technique originally reported in [Kirk83], and has been widely used in applications such as standard cell placement. It is based on the annealing principle of metal-

lurgy, where a crystal molecular stricture is formed when a metal is heated up to the melting point and then allowed to slowly cool.

In simulated annealing applications, the problem space is divided into a set of 'molecules'. Some form of 'movement' among the molecules is defined, and a measure of the 'energy' level of the system is defined. A cost function is created which evaluates the energy required for a particular move. This cost function is defined so that moves that do not lead to a solution require more energy than moves that do lead to a solution. The 'temperature' of the system is raised high enough to permit free random movement of the molecules. As the temperature is lowered, certain moves among the molecules are no longer made because they require a higher system energy level than the current temperature will allow. Once the temperature has reached a minimum, the system will have solidified (stabilized) into a solution system that is close to (if not the same as) the ideal solution.

In ADACC's state assigner, a molecule is defined as a state in the AFSM. Movement among molecules is defined as swapping the assignments of two randomly chosen states. The measure of energy is calculated with a localized cost function which can be determined independently for each state. This cost function is proportional to how good an assignment is for a particular state. The user's maximum transition and default transition timing constraints are used in the cost function to ensure that time critical transitions are assigned race-free assignments.

The annealing algorithm structure used in the state assigner is based on the placement annealer used in the TimberWolf placement and routing package [Sech85].

To help to explain this algorithm, a definition for the *order* of an assignment vector is required:

Definition 4.1 Order of a state assignment. The order of a particular assignment is the number of state variables used in each state vector.

Picking a move is done by randomly picking two assignments from the set of all possible assignments for the current assignment order. A random number generator is used to generate an integer between 0 and 2^{order} and then this integer is converted into a bit vector.

After two assignments are selected, a move is tested. This is done by first determining which states the chosen assignments belong to and then applying the local cost function to both states before and after the assignments are swapped. The difference of the cost functions are passed to an 'accept(C, T)' function which determines from the difference of the costs 'C' and the current system temperature 'T' if the move is to be kept or not. If this function returns false, then the assignments are swapped back (i.e., the move was rejected). There is the possibility that one or both of the chosen assignments are not used in any state in the FSM. If only one of the assignments is used in a state, then the cost function is only calculated for the state that uses this assignment. If both are not used, then the move is not performed.

The temperature is slowly lowered by the function 'update(T)', which lowers the temperature 'T' by multiplying it with the following function:

$$1 - \varepsilon, \text{ where } \varepsilon \text{ is } << 1,$$

This creates a smooth exponentially decaying temperature schedule. Another function, 'at_this_temp(T)', determines how many moves are to be performed at a particular temperature, which is set to vary with the size of the state machine. The stopping criteria is governed by a 'graph_is_melted(T)' function, which returns true if the temperature is above a specified minimum value.

Pseudo-code that describes the simulated annealing function is shown in Figure 4.3.3.

```
while (graph_is_melted(T))
{
    while (at_this_temp(T))
    {
        pick_move (assignment1, assignment2);
        C = make_move(assignment1, assignment2);
        if (accept(C, T)) keep_move();
        else reverse_move(assignment1, assignment2);
    }
    T = update(T);
}
```

Figure 4.3.3. Pseudo-code for simulated annealing

4.3.4. Annealing Cost Function

The cost function computes a measure of assignment desirability, based on assignments of adjacent states and user timing restrictions. The following definition will be used to describe the cost function:

Definition 4.2 Distance between state assignments. This is the number of bits in a state assignment that are different from the corresponding bits in another assignment. For example, the distance between the assignment [11001] and the assignment [01101] is two. Note that unit-distance assignments are by definition race-free.

The cost function evaluates the assignment of a state based on two criteria:

1. how close that assignment is to being unit-distance from the assignments of every adjacent state, and
2. the user timing restrictions associated with any transitions connected to that state.

The cost function associated with a state is calculated using the following equation:

$$cost = \sum_{i=1}^{N} \frac{(Di \times Ri)}{N}$$

The variables used in this equation are defined below:

Di: Distance between assignments of states connected with transition *i*

Ri: User timing factor of transition *i*

N: Number of transitions connected to the state

The cost function produces a real number ranging between 0.0 and 1.0. Here, a cost of 0.0 designates a perfect assignment for that state. The distances associated with transition arcs that have a user-specified maximum transition time are multiplied by a factor 'Ri' which is inversely proportional to this time. This increases the cost associated with arcs that are time constrained, which in turn causes the annealer to put more emphasis on giving good assignments to the states associated with these arcs. The factor 'Ri' is calculated by using the following equation:

$$Ri = K - (\frac{TTi}{DTT}) \times K + (\frac{TTi}{DTT})$$

Here, 'TTi' is the user-specified maximum transition time of the i'th transition, and 'DTT' is the user-specified default transition time for all transitions.

The form of the above equation was selected to give a linear function that varied from 1.0 to 'K' as 'TTi' varied. As 'TTi' approaches 'DTT', the value of 'Ri' approaches 1.0, which has no effect on the resulting cost function. This was done because a transition with a timing constraint equal to the default transition time is has, in effect, the same timing restriction as a transition with no timing constraint. However, as 'TTi' approaches zero, 'Ri' approaches the value of 'K', which has the effect of multiplying the cost function of this transition by 'K'. This increases the cost of tightly time-constrained transitions.

The value of 'K' was chosen to be equal to the number of transitions associated with the state that is being evaluated. This makes transitions with very low time constraints be more costly than the sum of all other costs associated with a state.

4.3.5. Annealing Temperature Scales

As the number of states in the graph increases, the amount of time spent annealing must increase to maintain good results. Here, this is achieved by increasing the margin between the starting and stopping temperatures and by increasing the number of moves performed at each temperature.

According to White [Whit84], the ideal stopping temperature of a simulated annealing system can be determined qualitatively by using a measure of the energy of the system. If E_0 represents the minimum energy of the system, E_1 is the energy of a system that is one move away from E_0, and M is the total number of moves originating from the system E_1, then we can write the following:

$T_0 = (E_1 - E_0) / \ln(M)$
where,
T_0 = the ideal stopping temperature of the system

M = (number of nodes)^2 = $(2^{order})^2$
$E_0 = 0$
E_1 = 1 / (maximum number of arcs attached to a node × order)
= 1 / (order × order)

This gives:
$T_0 = 1 / (2 \times order^3 \times \ln(2))$

This expression is used in the annealer to determine the final stopping temperature. For a system with 100 states, this gives a stopping temperature of 2.1×10^{-3} degrees. Very few uphill moves are made in the system at temperatures below this value.

The starting temperature of the system was determined empirically to be approximately 5 degrees. The method for doing this was to heat up test cases, run annealing, and determine the point at which almost all uphill moves are accepted. At this point, the system is hot enough to be fully 'liquefied'. However, if the temperature is lowered, fewer uphill moves will take place and the system will start to solidify.

In order to allow the system to reach steady state for each temperature, the number of moves allowed at each temperature must be increased as the size of the state machine increases. This function was determined empirically and was found to be:

$$\#iterations\ per\ temp\ increment = J \times 2^{order}$$

where 'J' is a constant. This expression makes intuitive sense since the total number of assignment

vectors in the system is 2^{order}, which implies that the number of iterations is proportional to the size of the machine.

4.3.6. Implementation of Algorithmic Cleanup Procedure

The cleanup procedure finds all adjacent state pairs that do not have race-free assignments and inserts additional states between them. The assignments of these additional states are chosen to differ from assignments of the original state pairs by only one bit, resulting in a race-free state transition. These additional states are chosen from the existing states (i.e., shared-row states), or can be newly added (transition) states.

Cleanup starts by sorting every non-race-free state transition in the graph so that the transitions that are the most difficult to complete are corrected first. As the cleanup procedure progresses, it uses up assignments that could be used to complete other transitions. Therefore it is advantageous to complete the transitions with the highest degree of freedom last.

The routine 'complete_arc()' is called for each one of these transitions. Complete_arc() tries to find a state or series of states that can be included in the transition in order to make it race-free. If 'complete_arc()' cannot find a series of transition or shared row states that satisfies the transition arc, then the order of the assignments is increased by one, and 'complete_arc()' is tried again. The pseudo-code for 'cleanup()' is supplied in Figure 4.3.4.

```
cleanup()
{
    list_of_arcs = find_all_bad_arcs();
    list_of_arcs = sort_arcs(list_of_arcs);
    for (each transition arc 'T1' in list_of_arcs)
    {
        worked = false;
        while (not worked)
        {
            worked = complete_arc(T1);
            if (not worked) increase order of assignment by 1
        }
    }
}
```

Figure 4.3.4. Pseudo-code for 'cleanup()'

As noted previously, sorting the bad transitions is done so that the cleanup routine works on the transitions that are the hardest to complete first. This notion of 'hardest to complete' is directly related to how *restricted* the transitions are. The following definitions describe what is meant by a restricted transition arc.

> *Definition 4.3* Restricted Assignment Vector. A restricted assignment vector has few unit distance neighbours that are not used by any state in the graph. The more unit distance neighbours that are used in states, the more restricted the assignment is.
>
> *Definition 4.4* Restricted Transition: A restricted transition is one in which the assignment vectors of the connected states are themselves restricted and the distance between these vectors is small. If the distance between the assignment vectors is small, then there are fewer ways to complete the arc, and therefore the arc is more restricted.

The 'complete_arc()' routine operates as follows. For each non-adjacent transition, an assignment is chosen that is unit distance away from the state on the transition with the most restricted assignment and reduces the distance to the other state. This ensures that the most constrained state adjacencies are satisfied first. The assignment is then checked to see if it can be used by checking to see if it is free. Free assignments can always be used, but if the assignment is already used in a state it can only be used if that state can be shared. If the assignment can be used, then a new state that uses this assignment is added in between these states. If the new state is adjacent to both states (i.e. unit distance away from both), then the transition is race-free, and 'complete_arc()' returns true. If the new state is

adjacent only to one state, then 'complete_arc()' is called recursively using the remaining non-adjacent transition. If the recursively called 'complete_arc()' returns true, then this new state is kept, otherwise, a new assignment and state are tried until all the appropriate assignments have been tried. This search for appropriate assignments is exhaustive. If an appropriate assignment cannot be found, then 'complete_arc()' returns false. In this way, transition states are added incrementally until a complete path between the original states is found. The pseudo-code for 'complete_arc()' is described in Figure 4.3.5.

```
int complete_arc(T1)
{
    if (arc_is_good(T1)) return (true);
    S1 = most restricted state attached to T1;
    S2 = least restricted state attached to T1;
    list_of_assignments = find_good_assignments(T1);
    for (each assignment 'A1' in list_of_assignments)
    {
        if (!(A1 is used in a state && cannot share that state))
        {
            if (A1 is used in a state) S3 = get_state(A1);
            else S3 = new_state(A1);
            T2 = insert_state (S1, S3, S2);
            if (complete_arc(T2)) /*then the transition is race-free*/
                return (true)
            else /* we must try another assignment */
                remove_state(S3); //
        }
    }
    return (false); /* could not complete the assignment */
}
```

Figure 4.3.5. Pseudo-code for 'complete_arc()'

An example of the operation of 'complete_arc()' for a typical transition is shown in Figure 4.3.6. In Figure 4.3.6a, state 'S0' has the assignment vector [1001], and is connected to state 'S1', which has the assignment vector [1110]. After the first call to 'complete_arc()', shown in Figure 4.3.6b, the state 'Sa' with the assignment [1101] is inserted between 'S0' and 'S1'. This creates a race-free transition between the states 'S0' and 'Sa', but the transition between 'Sa' and 'S1' is still non-adjacent. After the second call to 'complete_arc()', shown in Figure 4.3.6c, the state 'Sb' with the assignment [1100] is

added between the states 'Sa' and 'S1', which makes the entire transition race-free.

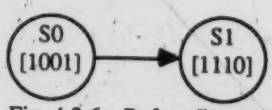

Fig. 4.3.6a. Before first call to `complete_arc()'

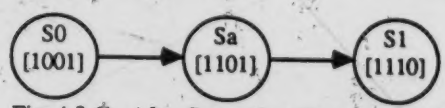

Fig. 4.3.6b. After first call to `complete_arc()'

Fig. 4.3.6c. After second call to `complete_arc()'

Figure 4.3.6. Example of 'complete_arc()' operation.

4.4. Removal of Combinational Hazards

As shown in Chapter 3, the next state and output equations generated from the assigned FSM likely contain combinational hazards that must be removed to ensure correct circuit operation. This section presents the implementation of the combinational hazard remover used in ADACC. The implementation of ADACC's next state and output equation generator will not be discussed here because it is a straight forward implementation of the formulas presented in section 3.4 of Chapter 3.

As stated in section 3.5 of Chapter 3, a circuit will be free of combinational hazards if all of the adjacent product terms are 'covered' by another product term.

Adding these additional terms has been implemented in a method of the equation class called 'cover_hazards()'. This method compares every term in the equation with every other term in the equation to try to find adjacent product terms. The method of the product class 'find_cover_term()' is used to determine if two terms are adjacent, and if so, returns a product term that covers all the adjacent input pairs of the original two terms. If the two terms are not adjacent, then it returns zero. A check is

performed to see if the term calculated by 'find_cover_term()' is equivalent to a term that is already in the equation. If this is true, then the new term is not added. The pseudo-code for 'cover_hazards()' is shown in Figure 4.4.1.

```
equation::cover_hazards(equation *eq)
/*****************************************************************************
* adds all prime implicant terms required to cover static & dynamic hazards
*****************************************************************************/
{
    for (every product term (prod1) in the equation eq)
    {
        for (every product term (prod2), with prod2 != prod1)
        {
            new_term = find_cover_term(prod1, prod2);
            if (new_term != 0 && there is no product term in this
                equation that is equivalent to new_term)
                add_product(new_term);
        }
    }
}
```

Figure 4.4.1. Pseudo-code for 'cover_hazards()'

The method 'find_cover_term()' makes sure that only one variable is different between the two product terms (not withstanding don't cares), and if so, creates a term that contains the values of the variables of both original terms, with the exception of the one term that is different. This new term thus 'covers' the two original terms. The pseudo-code that describes the operation of 'find_adjacent_term()' is shown in Figure 4.4.2.

Note that the array 'input_var[]' shown in the pseudo-code is a private array of the product class and holds the values of the input variables of a product term.

```
product* product::find_cover_term(product *prod1, product *prod2)
{
      product *cov_term;

      cov_term = new product;
      for (all input variables i in the terms prod1 and prod2)
      {
            if (prod1->input_var[i] == prod2->input_var[i])
                  cov_term->input_var[i] = prod1->input_var[i];
            else
                  if (prod1->input_var[i] != dont_care &&
                        prod2->input_var[i] == dont_care)
                        cov_term->input_var[i] = prod1->input_var[i];
                  else
                        if (prod1->input_var[i] == dont_care &&
                              prod2->input_var[i] != dont_care)
                              cov_term->input_var[i] = prod2->input_var[i];
                        else /* BOTH are different*/
                        {
                              if (there has been a previous variable
                                    that was different)
                              { /* then, they are not adjacent terms */
                                    delete cov_term; return (0);
                              }
                              else cov_term->input_var[i] = dont_care;
                        }
      }
      if (and only if there was only one variable that was different)
            return (cov_term);
      else  return (0);
}
```

Figure 4.4.2. Pseudo-code for 'find_cover_term()'

4.5. Detection of Essential Hazards

This section shows how essential hazards are detected in the assigned FSM. As stated in the background research section, essential hazards can be detected by simple inspection of the flow table by checking if the final state after one transition of an input is different than the final state after three transitions.

The only problem is that ADACC does not use a flow table representation of the FSM. The user's state machine is represented as a directed graph, which means that this algorithm is not directly

applicable. There are two solutions to this problem:

1. Build a flow table from the information in the graph.
2. Modify the essential hazard detection algorithm to apply to a graph.

The first solution requires the creation of an additional data structure. Although this does not appear to be a stumbling block on the surface, consider that the flow table of a 100 state machine with 10 inputs will require 2^{10} columns, and as many as 100 rows. A table this large is not trivial to create and manipulate. Therefore, the second solution was implemented.

Here, finding all the essential hazards that originate in a particular state is done by toggling every input variable three times, simulating the operation of the FSM with every toggle, and determining if the final state after three toggles is the same as the final state after one toggle. This is done for every stable total state, (i.e., for every combination of input and next state variables). The pseudo-code for this algorithm is presented in Figure 4.5.1.

```
hazards::find_all_hazards(state_graph *G)
{
    for (every state 'S' in the graph G)
    {
        for (every combination of input vectors 'V')
        {
            for (every variable 'var' in vector 'V')
            {
                let vector V2 = V
                toggle (variable 'var' in 'V');
                state1 = simulate(G, S, V);
                toggle (variable 'var' in 'V');
                state2 = simulate(G, S, V);
                toggle (variable 'var' in 'V');
                state3 = simulate(G, S, V);
                if (state1 != state3)
                    record_hazard(S, V2, var);
            }
        }
    }
}
```

Figure 4.5.1. Pseudo-code for 'find_hazards()'

Here, determining the final state after an input toggle is done with the 'simulate()' method of the state_graph class, which determines the final state of the FSM given a starting state and an input vector. Simulate() first determines which next state equation associated with the start state is true with the passed input vector and then traverses to the next state connected by that transition. This process is then repeated with the new state until a stable state is found (i.e. until none of the next state equations of a state are true using the passed input vector). As stated above, this solution does not require the enumeration of the full flow table and uses the existing state graph with no modifications.

When a hazard is detected in the above routine, it is recorded in a special hazard object which is saved until required later to set up the timing paths for logic synthesis. The following information is recorded for each hazard:

1. the starting state
2. the input vector
3. the input variable that is toggled to create the hazard
4. the state variable that changes during the first transition from the start state

This information is used to create timing restrictions for later use in the rule-based timing optimizer.

4.6. Implementation of Rule-Based Timing Optimization System

The rule-based timing optimization system is responsible for creating circuitry from the Boolean equations derived from the state_graph, and modifying this circuitry so that:

1. The circuitry only uses gates in the supplied technology library, and
2. The timing of the circuitry meets the required timing constraints.

The above two goals are accomplished by two sub-systems:

1. Technology Mapping
2. Timing Optimization

Technology mapping is responsible for creating an initial circuit consisting of generic AND, OR

and INV gates from the Boolean equations generated by the equation generator, and modifying this circuit so that it only uses the gates from a specific technology library. Timing optimization is applied after technology mapping. Its job is to modify the circuit to meet the timing requirements needed to remove essential hazard effects and to pass user timing restrictions.

The technology mapper is made up of the following parts:

1. Technology dependent gate library. This is is a collection of gates from the technology that is to be used in timing optimization.
2. Circuit creation mechanism. This is a set of methods that create the initial circuit from the Boolean equations created from the FSM.
3 Technology mapping rules. When applied, these rules substitute one or more generic gates with gates from the gate library.
4. Rule application mechanism. This is a series of routines that implement rule matching and application.

The timing optimizer is made up of the following parts:

1. Timing optimization rules. These rules are the topographical and gate delay rules presented in section 3.8.3 of Chapter 3.
2. Timing Restrictions. These are the user and essential hazard restrictions described in section 3.7 of Chapter 3. The object of timing optimization is to modify the circuit to meet these restrictions.
3. Rule application mechanism. This is the same mechanism used in the technology mapper.
4. Rule maintenance system. This system is a database of matched rules that apply to the circuit. It must be maintained because with each rule application, several of the matched rules will become obsolete and several other non-matched rules will have to be added.
5. Conflict Resolver. This system determines the best rule to apply out of all the rules in the rule maintenance system. It uses a timing analyser to determine which rule will improve the global circuit timing the most, and this rule is subsequently applied.
6. Timing Analyser. This subsystem is used by the conflict resolver to measure path delays in the circuit, and determine if the timing requirements that govern these paths are met.

Note that the technology mapper and the timing optimizer both share the same rule application mechanism.

The implementations of the technology mapper and the timing optimizer are described in the remainder of this Chapter.

4.6.1. Technology Mapper

The technology mapper takes the hazard-free Boolean equations generated from the user's assigned FSM and creates a generic circuit to implement these equations. After this circuit is created, it is mapped into the required technology by using a technology library and a set of mapping rules.

Technology mapping operates as follows:

1. Create a generic circuit to implement next state and output equations
2. For all gates in the circuit,
   2.1. Determine if that gate is in the technology library
   2.2. If it is not in the library find a mapping rule that can be matched to that gate
   2.3. Apply that rule
3. Repeat until all gates in the circuit are from the technology library

The implementation of each part of the above algorithm is described in the following sections.

4.6.1.1. Creation of Generic Circuitry

The initial circuit object is created using the Boolean equations that represent the next state and output variables of the assigned state machine. First, any required input or output ports are added to the circuit, then SOP representation of the Boolean equations are created using generic gates. The resulting circuitry is two-level, with a series of generic AND gates feeding into a very large generic OR gate. Inverters are used as required to invert inputs. An example of the circuitry created from a typical equation is shown in Figure 4.6.1.

None of the feedback paths are closed in the circuit by a connecting wire. Therefore, there is an input port and an output port assigned to each state variable. When the final netlist is created, these ports are connected which closes the feedback. In order to avoid confusion, state variable names used in the output ports have the '+' character appended to them. For example, if a state variable name was 'Q2', then the corresponding output state variable name would be 'Q2+', which represents the next state of that state variable.

Original Equation:

$$O1 = T*!Q2 + Q1*!Q2 + Q1*!T$$

Circuitry Created:

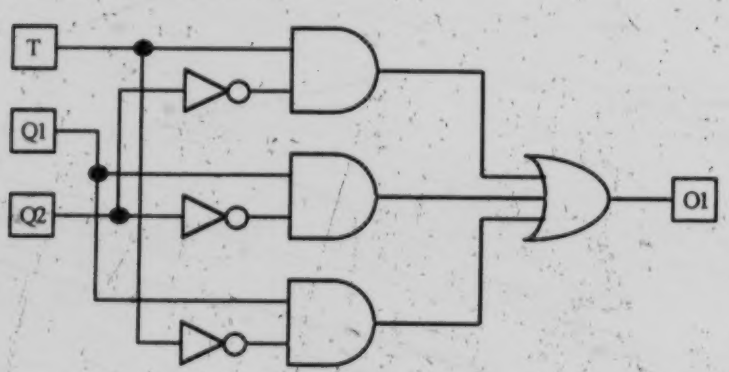

Figure 4.6.1. Example of a circuit created from a Boolean equation

4.6.1.2. Technology Dependent Gate Library

This library is a list of the gate objects that are available to optimization. The gate library contains AND and OR gates of various speeds, as well as several inverters of various speeds. These gates are all based on SSI level logic chips of the TTL 7400 series. A complete list of the gates in the library can be found in Appendix B.

The gate technology used in ADACC is restricted to one and two input gates. This is done to limit the number of gates in the library, and therefore to simplify the timing rules to a manageable level. It should be noted that timing optimization performance may be increased with the addition of gates with more than two inputs, but this improvement will not better demonstrate the idea of using rule-based optimization in ADACC.

4.6.1.3. Mapping Rules

The rules used in technology mapping have a single generic gate on the LHS and several two input gates from the library on the RHS. Only one rule is created for each generic gate. That is, there

is one and only one rule that matches a generic 2 input AND gate, and only one rule that matches a generic 7 input OR gate.

To simplify the mapping rules, the rules for large generic gates split these gates into two smaller generic gates and a single two input library gate. For example, the rule to map a seven input OR gate is written as follows:

(+, or7, a, b, c, d, e, f, g) → (+, or2_74ls32, (+, or3, a, b, c), (+, or4, d, e, f, g))

When this rule is applied the original 7 input generic gate is split into one 2 input library gate, one 3 input generic gate, and one 4 input generic gate. Here, any generic gates on the RHS are mapped later with the appropriate rule. This automatically creates a circuit with several gate levels in which all paths leading to a particular output are approximately the same level. This creates a good starting point for later timing optimization.

At present, the mapping rules create circuits that only use library gates from the 74ls gate series. These gates are shown in Figure 4.6.2.

	Gate:
1	74ls04 (inverter)
2	74ls08 (2 input AND)
3	74ls32 (2 input OR)

Figure 4.6.2. Library gates used in mapping rules

This generates a circuit with similar gates, which equalizes all the gate delays so that all path delays are approximately the same.

A complete list of the technology mapping rules is presented in Appendix D.

4.6.2. Rule Application

This section describes the mechanism for applying rules to a circuit. Rule application as presented here is used in timing optimization as well as technology mapping.

4.6.2.1. Rule Matching Mechanism

Before a rule is applied, it must be matched to a set of gates in the circuit, which is accomplished using a rule matcher. Rule matching is implemented with a graph equality checker. This checker takes a rule and a gate from the circuit and determines if the rule matches the circuit at that gate.

Here, the rule matcher is implemented with the routine 'match_rule()'. Match_rule() finds a portion of the circuit that is exactly the same as the LHS of the rule being considered. Once this portion of the circuit is found, then the addresses of the gates in this portion are entered in the 'match' pointers of the corresponding gates in the LHS.

4.6.2.2. Rule Application Mechanism

Rule application is implemented with the 'apply()' method of the rule class. This method finds the gates in the circuit matched to the LHS, deletes them, and adds new gates that are copies of the gates on the RHS. These new gates are then connected into the circuit, and matched with the gates on the RHS of the rule. Pseudo-code for the 'apply()' method is presented in Figure 4.6.3.

```
int rule::apply(circuit *cir1)
{
    apply_delete(cir1); // delete the original nodes in the circuit
    apply_add(cir1); // add replacement nodes to the circuit
    apply_connect(cir1); // connect up the new nodes

    return (0);
}
```

Figure 4.6.3. Pseudo-code for 'apply()'.

'Apply()' calls three other methods, 'apply_delete()', 'apply_add()', and 'apply_connect()'. Apply_delete() finds the gates in the circuit that correspond to all the gates on the LHS of the rule and

then deletes these gates from the circuit. The wires that are connected to these gates are deleted also. 'Apply_add()' creates a copy of every gate on the RHS, saves the addresses of these gates in the match pointers of the appropriate RHS gates, then adds these gates to the circuit. 'apply_connect()' then connects up the freshly added gates so that they implement the same circuit as the RHS.

In addition to 'apply()', the rule class contains the method 'un_apply()'. The purpose of this second method is to undo the effects of a recently applied rule. This is implemented in much the same way that apply is implemented, with the exception that the gates that are deleted correspond to the RHS of the rule (as opposed to the LHS), and the gates that are added correspond to the LHS of the rule.

4.6.3. Timing Optimizer

The purpose of timing optimization is to adjust path delays so that timing restrictions required by the user and required to hide essential hazards are met.

Timing optimization uses the same rule matching and application methods used in technology mapping, but requires a much more extensive rule search and maintenance system. Rule searching requires a timing analyser and a conflict resolver to choose the appropriate rule from the rule database, while rule maintenance is required to keep track of rules that are matched to the circuit.

Timing optimization starts by examining all the timing restrictions in order to find the paths in the circuit that do not meet their required timing. The path with the worst timing is then selected, and all gates along it are found and put into a list. The rules matched to these gates are retrieved from the rule maintenance system and are checked with the 'resolve_conflict()' routine. This routine determines which rule creates the greatest improvement in circuit timing, after which that rule is applied to the circuit. The rule maintenance system is updated after the rule is applied. This process is repeated until all paths in the circuit meet the required timing, or until the system cannot find any more downhill rules to apply. The pseudo-code for timing optimization is shown in Figure 4.6.4.

```
int timing_op(circuit *circ, rule_base *timing_rules, list_of_restrictions *restric)
{
    path *p1;
    rule *best_rule;
    gate *gate;
    datapSLList gate_list;

    for (every path 'p1' who's timing constraints are not satisfied)
    {
        gate_list = find_all_gates_along(p1);
        best_rule = 0;
        for (every gate 'gate' in 'gate_list')
        {
            // get the rules that are matched to gate:
            rule_list = gate->private_rule_list;
            for (every rule 'rule' in 'rule_list')
                best_rule = resolve_conflict(best_rule, rule, circ, restric);
        }
        if (best_rule != 0)
        {
            // then we have a rule to apply!!!
            best_rule->apply(circ, orig_rb, -1);
            // now update rule maintenance system:
            delete_outdated_rules(circ, best_rule);
            add_new_rules(circ, timing_rules, best_rule);
        }
        if (no more rules can be found to apply) return (10); //optimization failed
    }
    return (0);
}
```

Figure 4.6.4. Pseudo-code for timing optimization

The implementation of the rule maintenance system, the conflict resolver, and the timing analyser are described below:

4.6.3.1. Rule Maintenance System

During timing optimization, there are a large number of rules that can be matched to any particular set of gates. Initially, all possible rules are matched before any of the rules can be chosen. After a rule is applied, the subset of the rules matched to gates that no longer appear in the circuit must be removed, and new rules will have to be matched to the newly-added gates. This means that the set of matched rules must be updated after each rule application.

This is done with the rule maintenance system. This is a database of matched rules plus mechanisms for keeping the rules up to date. It is implemented by associating with each gate in the circuit a private rule list that contains all the matched rules that can affect that gate if any of those rules were to be applied. This localizes the matched rules to individual gates in the circuit.

When a rule is applied, the rules that are made obsolete by the application will be in the private rule lists of the gates that are deleted. Therefore, the problem of deleting the obsolete rules is reduced to finding all the rules that are matched to gates that are to be deleted in a rule application. This is implemented with the routine 'delete_outdated_rules()', which deletes all rules that will become obsolete with the application of a particular rule. This method checks the private rule list of every gate in the circuit for any rules that are matched to the gates that will be deleted if that rule is applied. If any rules are found they are deleted from the private rule lists. The pseudo-code for 'delete_outdated_rules()' is presented in Figure 4.6.5.

```
int delete_outdated_rules(circuit *c1, rule *rule1)
/*****************************************************************************
* this routine deletes all of the rules that will become obsolete with the
* application of the rule rule1.
*****************************************************************************/
{
    gate *rule_gate, *cir_gate;
    rule_base *rb; rule *rule2;

    for (each gate 'cir_gate' in c1)
    {
        list = cir_gate->private_rule_list;
        for (each rule 'rule2' in the list)
        {
            if (rule2 is matched to at least one gate that is also
                matched to rule1) // then rule1 is obsolete
                list->delete_rule(rule1);
        }
    }
    return (0);
}
```

Figure 4.6.5. Pseudo-code for 'delete_outdated_rules()'

After a rule is applied, the private rule lists of the newly-added gates need to be created. In addition, any of the new rules in these lists that affect old gates need to be added to the private rule lists of these old gates as well. This is implemented with the routine 'add_new_rules()'. This method first finds and matches rules to the new gates, then it determines if there are any old gates that are affected by these new rules, and if so, adds copies of these rules to those gates. Matching rules to the newly-added gates is done by calling the method 'match_rules()' for every new gate. This method generates a set of rules that can be applied to a gate in the circuit. This rule set is then put into the private rule list of the new gate. Copying these new rules to any old gates in the circuit that they effect is done by calling the method 'distribute_rule()' for every newly matched rule in these new gates. Distribute_rule() finds all the old gates that are pointed to by the 'match' pointers on the LHS of a new rule, and then copies that rule into the private rule lists of these old gates. The pseudo-code for 'add_new_rules()' and 'distribute_rule()' is shown in Figure 4.6.6 and Figure 4.6.7.

```
int add_new_rules(circuit *circ, rule_base *timing_rules, rule *rule1)
/*****************************************************************************
* This method sets up the private rule lists of all the gates that are either
* added or otherwise affected with the application of rule1.
*****************************************************************************/
{
    gate *rule_gate, *cct_gate;
    rule *rule2;

    match_new_gates(circ, timing_rules);
    for (each gate 'rule_gate' on the RHS of rule1)
    {
        cct_gate = rule_gate->match;
        cct_gate->private_rule_list = match_rules(timing_rules, cct_gate);
    }
    // Now to distribute all those new rules:
    for (each gate 'rule_gate' on the RHS of rule1)
    {
        cct_gate = rule_gate->match;
        list = cct_gate->private_rule_list;
        if (list != 0)
        {
            for (every rule 'rule2' in 'list')
                distribute_rule(circ, rule2);
        }
    }
    return (0);
}
```

Figure 4.6.6. Pseudo-code for 'add_new_rules()'

```
int distribute_rule(circuit *c1, rule *rule1)
/*****************************************************************************
* This method takes this rule and adds a copy of it to all of the gates
* that it modifies if it were to be applied.
*****************************************************************************/
{
    gate *rule_gate, *cir_gate;

    for (every gate 'rule_gate' on the LHS of rule1)
    {
        cir_gate = rule_gate->match;
        list = cir_gate->private_rule_list;
        add a copy of this rule to list;
    }
    return (0);
}
```

Figure 4.6.7. Pseudo-code for 'distribute_rule()'

4.6.3.2. Conflict Resolver

The purpose of the conflict resolver is to select the next rule to apply. Here the goal is to find the rule that generates the greatest improvement of overall circuit timing. This is achieved by evaluating rules to determine how they improve the timing of all the paths in the circuit.

The conflict resolver is implemented with the routine 'resolve_conflict()'. This routine evaluates two rules, and returns the rule that creates the largest improvement of the global timing of the circuit. If neither of the rules improve the timing, then zero is returned. Here, another routine called 'global_evaluate_rule()' is used to evaluate the suitability of a rule. This routine returns a cost value that represents the improvement/degradation of global timing of the circuit in nanoseconds if the passed rule were to be applied. If the timing is degraded, a cost greater than zero is returned, and if timing is improved, a cost less than zero is returned. Therefore, 'resolve_conflict()' simply returns the rule with the smallest negative cost. If the costs of both rules are positive, however, neither rule is returned. Pseudo-code for 'resolve_conflict()' is presented in Figure 4.6.8.

'Global_evaluate_rule()' determines if the timing restrictions are met, and if not, how close they are to being met. Then it determines how close the timing restrictions are to being met *if the passed*

```
}
rule* resolve_conflict(rule *r1, rule *r2, list_of_restrictions *lr, circuit *c1)
{
    int a_min, a_max, a1, b1, a2, b2,
    cost1 = 0, cost2 = 0, ret;

    cost1 = c1->global_evaluate_rule(lr, r1);
    cost2 = c1->global_evaluate_rule(lr, r2);
    if (cost1 <= cost2 && cost1 <= 0) return (r1);
    if (cost2 < cost1 && cost2 <= 0) return (r2);
    return (0); // no uphill rules accepted
}
```

Figure 4.6.8. Pseudo-code for 'resolve_conflict()'

rule was to be applied. This is done using the routine 'predict_timing()', which determines the timing of a path both before and after a rule is applied and determines if either of these times violate the required timing of that path. Then the rule is un-applied in order to put the circuit back the way it was.

Predict_timing() returns two values, one representing how well the path meets its timing constraint before the rule is applied, and another representing how well the path meets its timing constraint after the rule is applied. If the values are negative, they represent the timing margin in nanoseconds that the constraint is met by. If the values are positive, they represent the time in nanoseconds that the path must be adjusted in order to meet the constraint. Therefore, these values can be compared to determine how effective the rule was in helping the path meet the timing constraint. 'Predict_timing()' uses the routine 'check_timing()' described in the Timing Analyser section of this Chapter to perform timing tests. Pseudo-code for 'global_evaluate_rule()' is shown in Figure 4.6.9.

```
int circuit::global_evaluate_rule(list_of_restrictions *lr, rule *rule1)
{
    int before, after, before_sum, after_sum;
    timing_restriction *tr;

    for (every restriction 'tr' in the list of restrictions 'lr')
    {
        predict_timing(before, after, tr, rule1);
        after_sum += after;
        before_sum += before;
    }
    return (after_sum - before_sum);
}
```

Figure 4.6.9. Pseudo-code for 'global_evaluate_rule()'

4.6.3.3. Timing Analyser

The timing analyser is responsible for determining if the timing restrictions are satisfied in the circuit. Therefore, the timing delays of the original paths need to be found, and the *required* timing delays of these paths need to be determined. Comparing the timing delays of the original paths with the required delays can then be used to determine if the original paths meet the required timing restriction.

As explained in Chapter 3, the timing analysis performed here does not need to determine the path that is active in a particular total state; only the minimum and maximum path delays are required in order to guarantee correct circuit timing. Therefore, the timing analyser does not use the total state of the circuit to determine the speed of the path.

The timing analysis is performed by the 'check_timing()' routine. This routine determines the minimum and maximum time delay of a particular path in the circuit, and the required minimum and maximum times that the path must lie between to be acceptable. The pseudo-code that describes 'check_timing()' is presented in Figure 4.6.10.

'Time_path()' is used to time gate delay paths in the circuit and returns both the minimum delay, and the maximum delay of a path. The 'path_descriptor' structure of the timing restriction object only holds the names of the input and output ports on the path of interest. Note that there could be several

```
int circuit::check_timing(timing_restriction *r1, int &min, int &max, int &req_min, int &req_max)
{
        time_path(min, max, r1->path_descriptor);
        find_req_timing(req_min, req_max, r1);
}
```

Figure 4.6.10. Pseudo-code for 'check_timing()'

sequences of gates between these input and output ports. 'Time_path()' goes through every one of these sequences, summing the gates along them. Once all the gate sequences have been traversed, the shortest one and the longest one are determined and their times are returned.

The 'find_req_timing()' routine determines the required timing of the restricted path. That is, it determines what the timing of the path *should* be in order to be acceptable. This is done by examining the timing restriction that governs that path. If this restriction is a user-defined default transition time restriction or a user-defined maximum transition time restriction, then the required delay of the path is simply a constant time saved in the timing restriction. However, if this restriction is an essential hazard restriction, then determining the required timing must be done by measuring another path in the circuit (i.e. the secondary path described in Chapter 3). The delay of this path is the required timing of the original path. The pseudo-code of 'find_req_timing()' is shown in Figure 4.6.11.

```
find_req_timing(int &req_min, int &req_max, timing_restriction *tr)
/*****************************************************************************
* determines the timing that this path is SUPPOSED to have.
*****************************************************************************/
{
    int min, max;
    if (r1 is a trans_time restriction)
    {
        req_max = r1->trans_time;
        req_min = invalid;
    }
    if (r1 is a default trans time restriction)
    {
        req_max = r1->default_trans_time;
        req_min = invalid;
    }
    if (r1 is an essential hazard restriction)
    {
        time_path(r1->secondary_path_descrip, min, max);
        req_min = max;
        req_max = invalid;
    }
}
```

Figure 4.6.11. Pseudo-code for 'find_req_timing()'

4.6.4. Post Timing Optimization

After timing optimization has been completed, the circuit is in its final form. At this time, a net-list for the circuit is generated so that simulation can be performed in order to verify correct operation. Chapter 5 discusses the results of ADACC's state assigner, and Chapter 6 discusses the simulation results of some circuit examples synthesized with ADACC.

5. State Assignment Test Results

ADACC's state assigner was tested independently from ADACC as a whole. This chapter presents test results for two different configurations of the state assigner. In the first configuration, the annealing portion of state assignment was run using the theoretical minimum number of state variables that could be used to fully assign each test machine. In the second configuration, annealing was run using one additional state variable in order to determine if the assignment could be improved if the annealing space was increased. Both configurations were tested using seven state machines ranging from 4 to 100 states. The tests were performed on a Sun 3/260 workstation with a 68881 floating point co-processor and 16Mb of main memory.

The test results using the minimum number of state variables are tabulated in Table 5.1 and the test results using one additional state variable during annealing are tabulated in Table 5.2.

Test State Machine	1	2	3	4	5	6	7
Original Number of States	3	15	16	33	35	76	100
Original Number of Transitions	3	22	19	46	49	102	127
CPU Time Used	71 Sec	520 Sec	230 Sec	2300 Sec	1300 Sec	6000 Sec	5100 Sec
Non-Race-Free Trans. After Annealing	1	4	0	2	8	11	19
No. Added Transition States	1	9	0	5	15	22	38
No. Shared States	0	1	0	0	1	1	3
No. Used State Variables	2	5	4	6	8	8	9
No. State Vars. Above Minimum	0	1	0	0	2	1	2

Table 5.1. Assignment performance with minimum number of state variables

Test State Machine	1	2	3	4	5	6	7
Original Number of States	3	15	16	33	35	76	100
Original Number of Transitions	3	22	19	46	49	102	127
CPU Time Used	400 Sec	470 Sec	710 Sec	3900 Sec	5700 Sec	13500 Sec	16600 Sec
Non-Race-Free Trans. After Annealing	2	2	0	2	9	12	13
No. Added Transition States	3	4	0	2	32	21	26
No. Shared States	0	0	0	0	1	0	0
No. Used State Variables	3	5	5	7	8	10	9
No. State Vars. Above Minimum	1	1	1	1	2	3	2

Table 5.2. Assignment performance with one additional state variable

Test machine 2 is an asynchronous version of a synchronous design of a VMEbus arbiter [Moto85], which is currently being used in a product developed by a local electronics company.

Test machine 4 is a state machine implementation of the circuit described in [Borr87]. This circuit is an interface transducer between the Intel Multibus and a custom MOS integrated circuit.

The results in Table 5.1 show that percentage of bad transition arcs remaining after annealing was on average 15%. The cleanup procedure added as many transition/shared states as required to make the assignment race-free, which was on average 30% additional states. The order (see Definition 4.1) of the state assignments after cleanup was low in all cases. Only 2 additional state variables were added dur-

ing the cleanup of test machines 3 and 4. Test machines 5 and 6 only required one additional variable, while the rest of the machines did not require any.

Comparing Table 5.1 and Table 5.2 shows some advantages and disadvantages to including an additional state variable during annealing. All of the runtimes in Table 5.2 are larger than the times reported in Table 5.1, which reflects the increased time required to anneal the larger search spaces. Machines 1 and 2 do not seem to benefit at all from the inclusion of the additional variable; the order of the assignment of machine 2 was increased from 4 to 5, while machine 1 required two more transition states as well as a larger order. Although machine 3 did not require any more state variables, the number of required transition states was increased from 15 to 32, which is a dramatic decrease in the potential speed of the resulting circuit. However, machines 4, 5, 6, and 7 required fewer transition states with the second configuration, although machines 6 and 7 respectively required two and one more state variables to complete their assignments.

All of the assignments produced by state assignment were race-free, which is the primary requirement of ADACC's state assigner. In the first configuration, the annealer managed to correctly assign 85% of the transitions of a typical state machine, while the cleanup procedure required 30% additional transition/shared states, and at most 2 additional state variables to complete the assignment.

The second configuration required more time to run and although it reduced the number of transition states in some of the test state machines, it either increased or did not change the number of state variables required by assignment. Therefore the second configuration was considered to be a failure because of the increased number of required state variables. Currently, ADACC uses the first state assignment configuration.

6. ADACC Test Results

6.1. Overview

The performance of ADACC as a whole has been tested by having it synthesize a variety of FSMs. These range from a simple flipflop to a VME bus arbiter.

This chapter presents ADACC's results on four test cases. For each test case, the original FSM is shown, the state assignment produced by ADACC is examined, and the performance of rule-based optimization is analysed. The resulting circuitry is then simulated using the SILOS logic simulator to show that the circuits operate correctly without any visible hazards.

6.2. Test Machine 1 - Lab1 of EE535

The first machine that was synthesised using ADACC was a simple FSM that is used in the first lab of EE535, an undergraduate digital logic course taught at the University of Alberta. In this lab, students were asked to design a simple asynchronous circuit that implemented an inhibited-toggle flip-flop, as described in problem 27 B of [Roth79]. This flip flop had three inputs, 'i0', 'i1', and 't', and one output, 'out'. It was to be designed so that 'out' would change states if i0 = 1, and t changed from a 0 to a 1, or if i1 = 1 and t changed from a 1 to a 0. The flip flop was not to change state under any other conditions. A state machine that implements this operation is shown in Figure 6.1, and was used as the starting point of ADACC. No user timing constrains were included with this design.

This FSM was given the state assignment shown in Figure 6.2 by ADACC. ADACC used two state variables, Q1 and Q2, and managed to fully assign the FSM without any transition or shared states. Note that every transition in Figure 6.2 is race-free.

After the assignment was complete, the next state and output equations for the circuit were generated. These equations were first reduced, then redundant terms were added to take care of combinational hazards. This resulted in the Boolean equations shown in Figure 6.3.

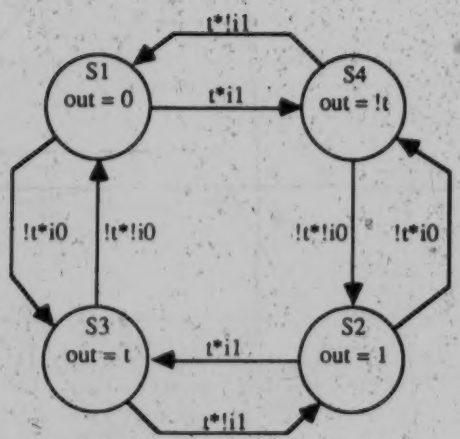

Figure 6.1. Starting FSM for inhibited-toggle flip-flop

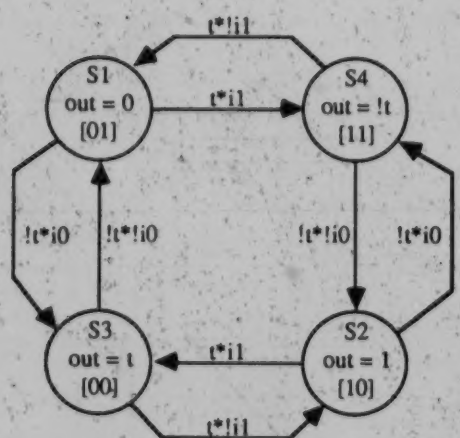

Figure 6.2. FSM for inhibited-toggle flip-flop after assignment

Q1+ = t*i1*Q2 + !t*Q1 + t*!i1*!Q2 + !i1*Q1*!Q2
Q2+ = t*Q2 + !t*!i0*!Q1 + !t*i0*Q1 + !i0*!Q1*Q2 + i0*Q1*Q2
out = !t*Q1 + t*!Q2 + Q1*!Q2

Figure 6.3. Next state and output equations

These equations proved to be exactly the same as equations derived by hand by students in the lab and an experienced hardware designer.

After the equations were developed, the state_graph was checked for essential hazards. Several were discovered, all related to the input 't'. The essential hazards detected by ADACC are shown in table 6.1.

Table 6.1. Essential hazards detected by ADACC

Input Variable	State Variable	Start State	Next State
t	Q2	s1	s3
t	Q1	s1	s4
t	Q2	s2	s4
t	Q1	s2	s3
t	Q1	s3	s2
t	Q2	s3	s1
t	Q1	s4	s1
t	Q2	s4	s2

The hazards shown in Table 6.1 were verified by hand and it was determined that ADACC found all the essential hazards in the circuit.

The next step was to run the above equations through the technology mapper to arrive at a starting point for rule-based timing optimization. This created the circuit shown in Figures 6.4a, 6.4b, and 6.4c.

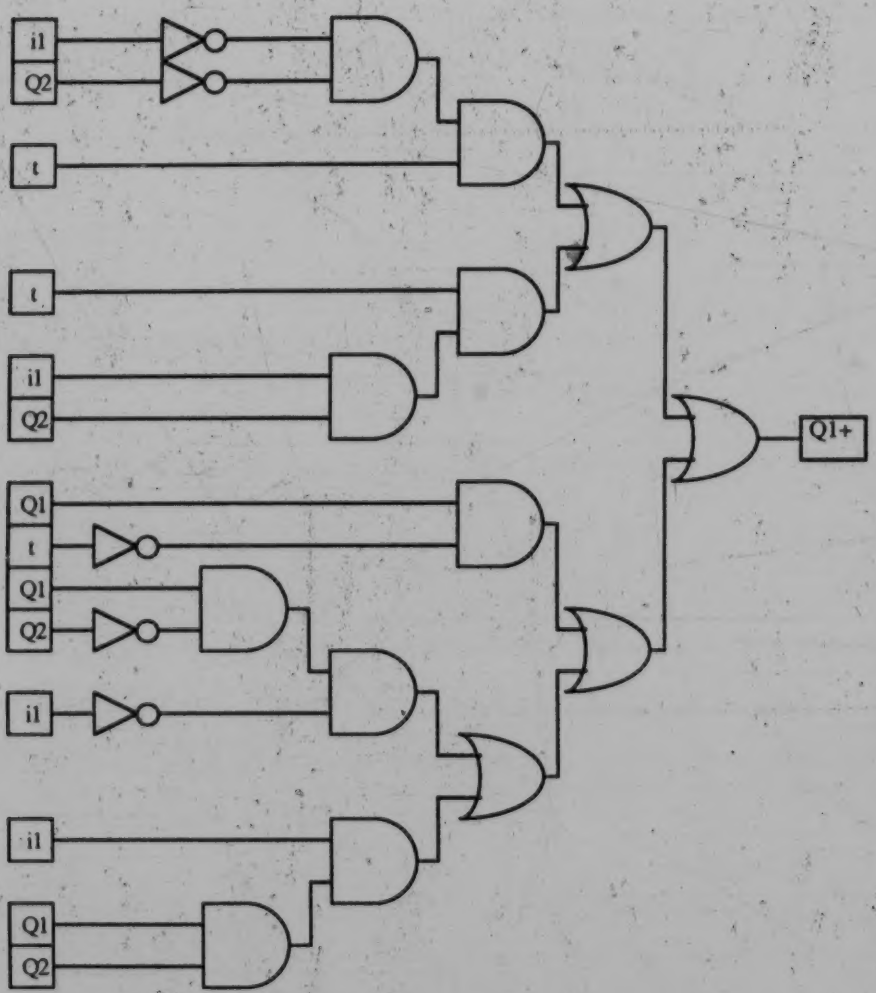

Figure 6.4a. Schematic of variable 'Q1+' after technology mapping

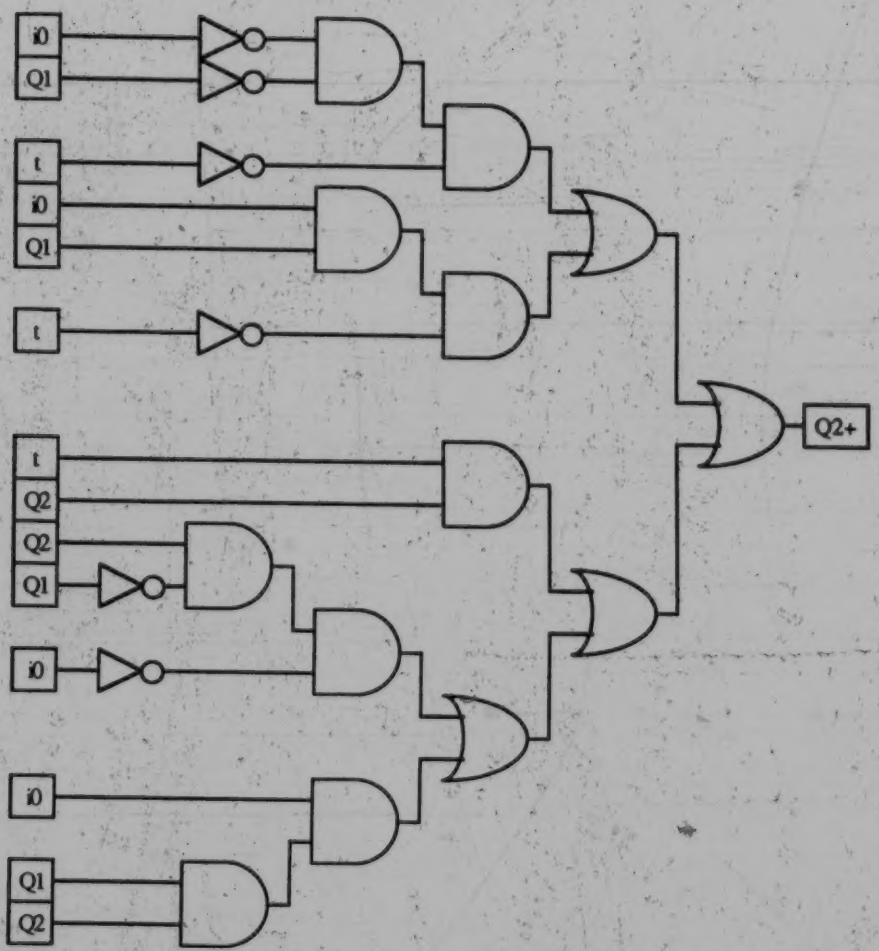

Figure 6.4b. Schematic of variable 'Q2+' after technology mapping

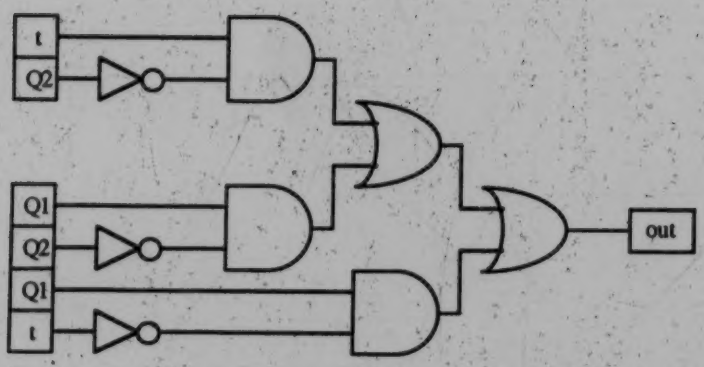

Figure 6.4c. Schematic of variable 'out' after technology mapping

On the first run of timing optimization, the system determined that the starting circuit did not violate any hazard timing restrictions, which means that the circuit shown in Figure 6.4 should work without any timing problems. This was verified with the SILOS logic simulator, using the two test cases shown in the timing diagrams shown in Figure 6.5. These diagrams show the waveforms of the input signals (t, i0 and i1), the waveforms of the next state variables (Q1 and Q2), the output, and the *expected* output.

The tests shown in Figure 6.5 show that the 'out' variable behaved exactly as expected. The 'out' signal was delayed approximately 50ns because of network delays in the circuit. None of the state variables or the output variable exhibited any spikes or glitches at any time during the tests, which shows that the circuit operated without any detectable hazards.

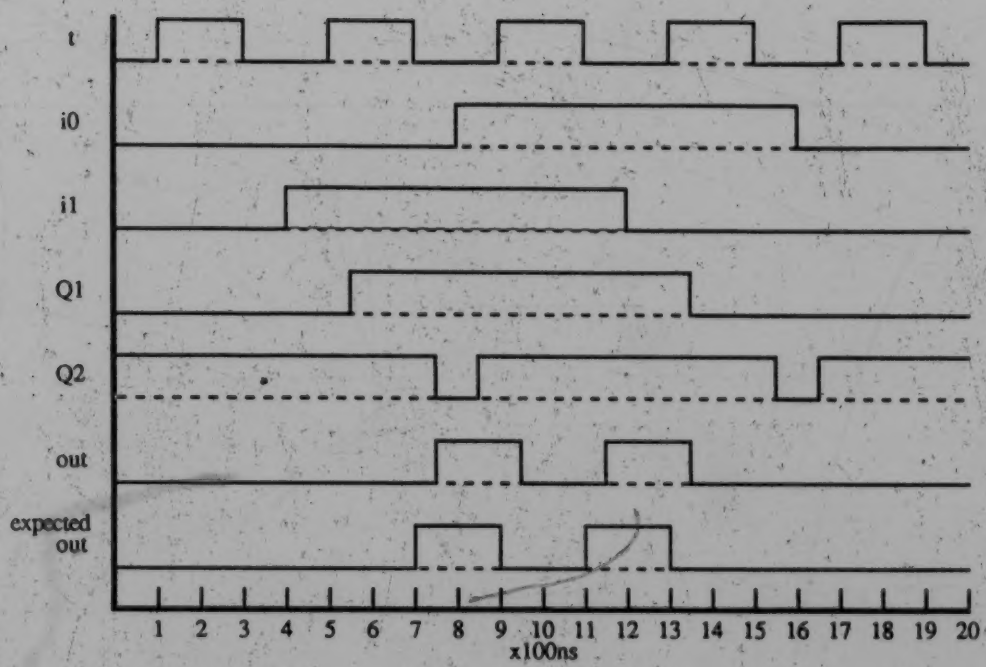

Figure 6.5. Tests results of SILOS simulation of original circuit

Because this logic simulation test does not show ADACC's ability to adjust circuit timing to eliminate essential hazard effects, the circuit was modified to uncover an essential hazard by delaying the input 't'. Specifically, the path from variable 't' to the state variable 'Q1' was delayed 72ns by including eight inverters as shown in the schematic in Figure 6.6. This increased the secondary path delay between t and Q1, and therefore should cause an essential hazard when the circuit switches between state S1 (01) and state S3 (00). This circuit was then simulated using SILOS, and the results of the simulation are shown in Figure 6.7. In this test case, the outputs and state variables start showing spikes and oscillations immediately after the state transition from state S1 to S3. The circuit did not recover from these problems.

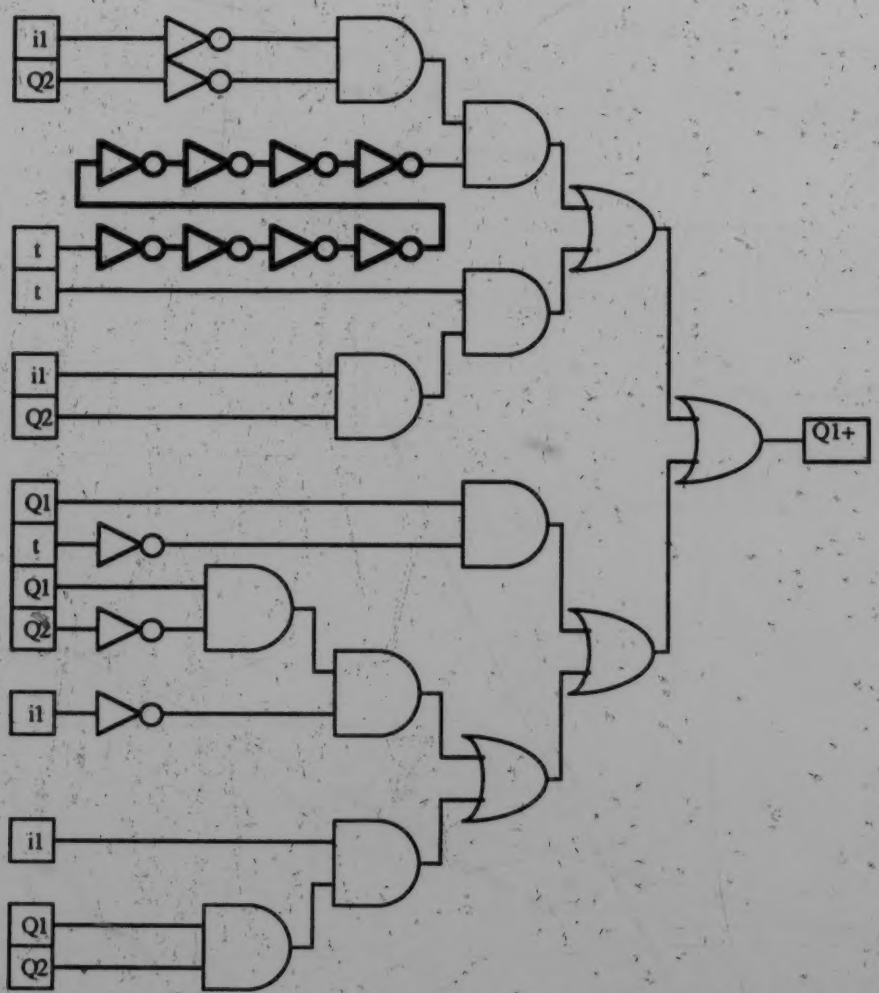

Figure 6.6. Schematic of variable 'Q1+' showing additional inverters

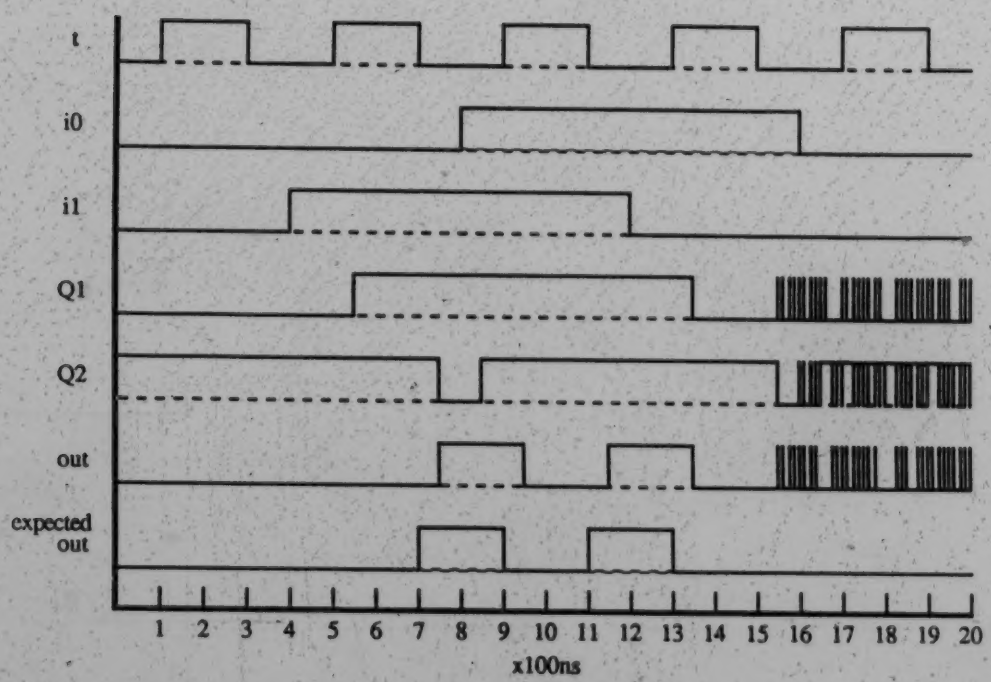

Figure 6.7. Test results of SILOS simulation showing essential hazard

After the inverters were added to the circuit, ADACC's timing optimization was used to eliminate the effect of the essential hazard. The timing problem was detected by ADACC, and was solved by applying only two topographical rules. These rules are listed in Figure 6.8.

1: out = (#, or2_74ls32, z, (#, or2_74ls32, x, y)) → out = (#, or2_74ls32, x, (#, or2_74ls32, z, y))

2: out = (#, or2_74ls32, (#, or2_74ls32, x, y) , z) → out = (#, or2_74ls32, x, (#, or2_74ls32, z, y))

Figure 6.8. Rules used to eliminate essential hazard

Rule 1 in Figure 6.8 was used to increase the delay of the primary path from t to Q2+, and rule 2 was used to decrease the secondary path between t and Q1+. Figure 6.9a shows how rule 2 was used to decrease the secondary path delay by removing an OR gate in the longest path between t and Q1+. This reduces the delay from 118ns down to 101ns. Note that this is the same path that had the 8 inverters inserted into it to uncover the essential hazard. Figure 6.9b shows how rule 1 was used to increase the

primary path delay by adding an additional OR gate in the shortest path between t and Q2+. Here this path was increased from 3 gate delays (46ns) to 4 gate delays (63ns). Each of these rules was used only once. Note that no gate delay rules were required to meet the timing restrictions. Note that the eight inverters added to uncover the hazard were not eliminated by optimization. This is because all rules using inverters were removed from the rule base to show that the timing problem could be eliminated by adjusting the topology of the circuit and not by simply removing the additional inverters.

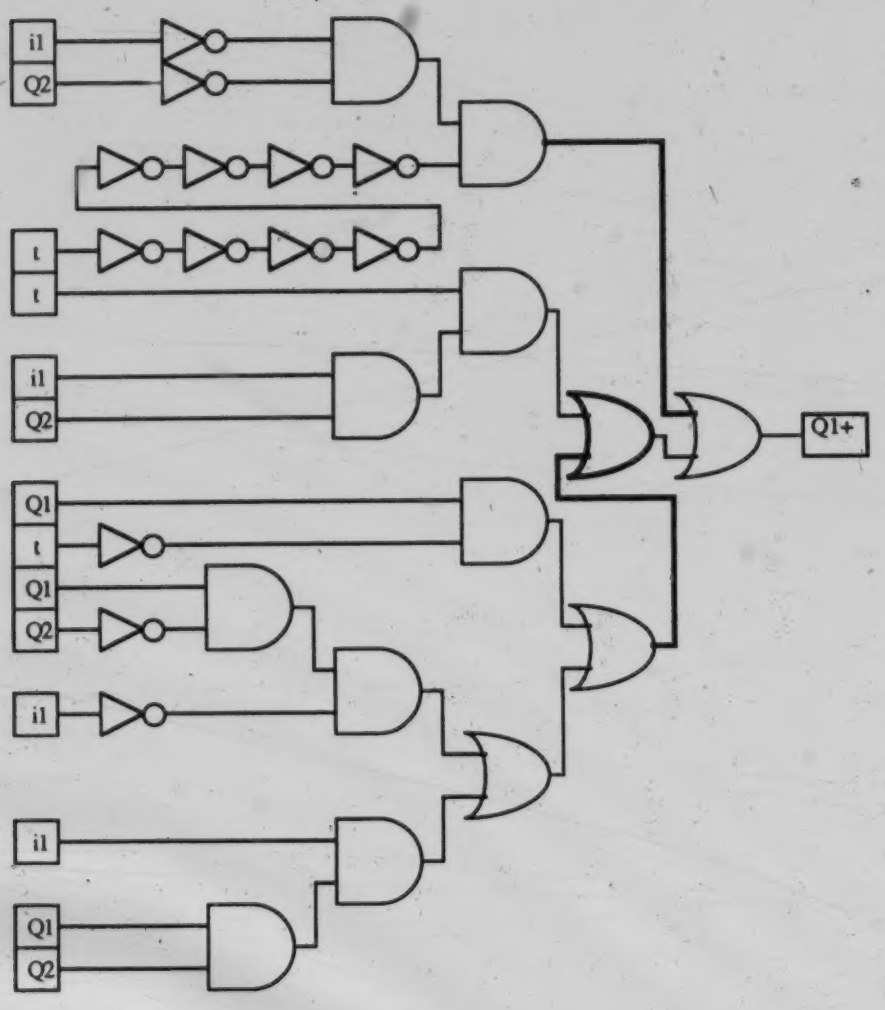

Figure 6.9a. Schematic of state variable 'Q1+' after optimization

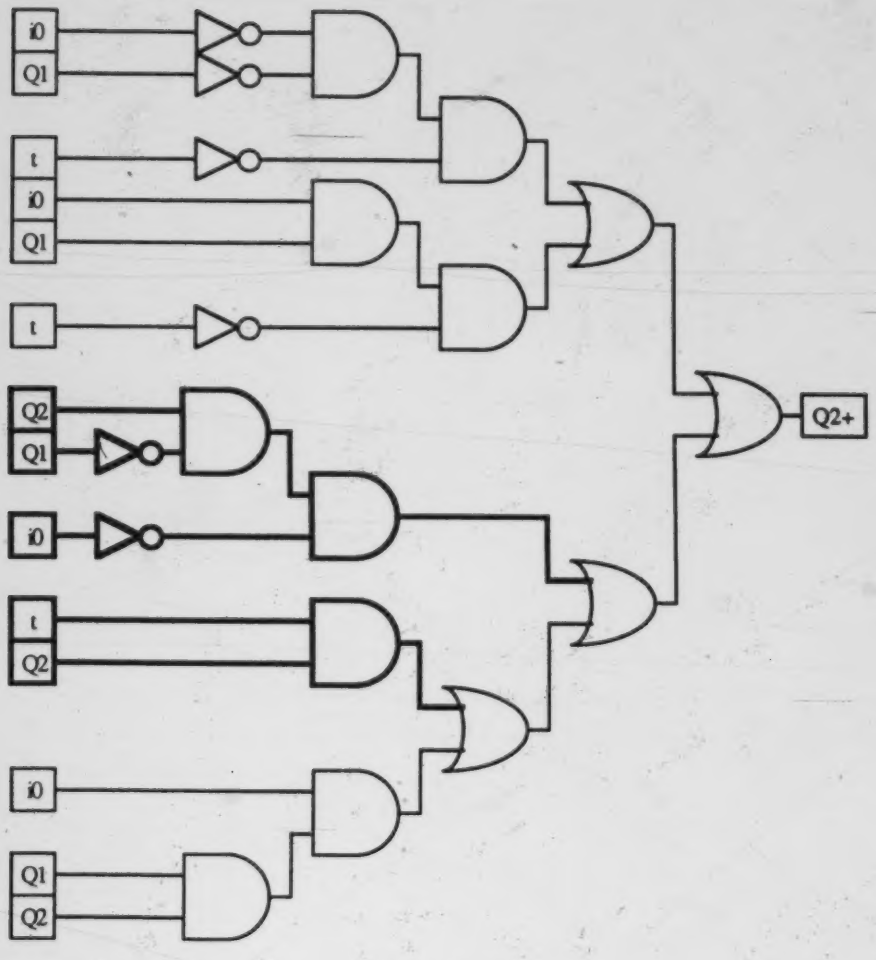

Figure 6.9b. Schematic of state variable 'Q2+' after optimization

The circuit created by timing optimization was again simulated using SILOS. The results in Figure 6.10. show that the circuit no longer exhibits the spikes and oscillations, and operates perfectly. This shows that ADACC corrected the essential hazard timing problem using *only* topographical rules to create a working circuit.

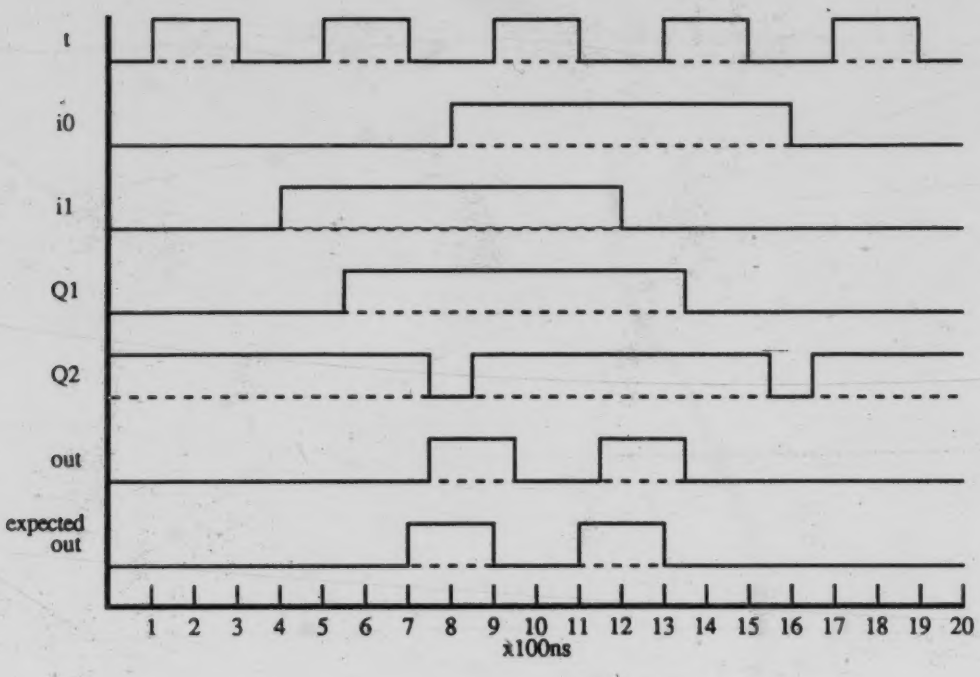

Figure 6.10. Test results of final circuit using SILOS
Essential hazard has been hidden

6.3. Test Machine 2 - VME Bus Arbitrator

The next AFSM synthesised using ADACC was an asynchronous version of a VME bus arbiter [Moto85]. The original synchronous version of this arbiter is currently used in a product manufactured by a local electronics company. The purpose of the arbiter is to allow access of the VME bus to only one peripheral board at a time. In addition, the arbiter is responsible for forcing a board to release control of the bus if another board with a higher priority requests it.

There are three priority levels used in this arbiter. The highest level is level 1, and the lowest level is level 3. Each board on the VME bus that is capable of becoming a bus master is assigned one of these three levels. Note that this arbiter is a slightly simpler version of the standard VME bus arbiter, as it only allows three priority levels instead of four.

Arbitration begins with one or more peripheral boards requesting the bus. Each peripheral board is assigned one of three bus request signals, 'br1', 'br2', or 'br3' which are daisy-chained from board to board. In response to one of these three bus request signals, the arbiter asserts one of three bus grant signals, 'bg1', 'bg2', or 'bg3' to tell the board that won the arbitration that it is free to use the bus. This board then asserts the 'bbsy' signal to inform all devices connected to the bus that it is in use. If a bus request is asserted that has a higher priority than the board currently using the bus, then the arbiter asserts the 'bclr' signal to inform the lower priority board to give up the bus as soon as possible. The arbiter also uses a 'reset' signal, which is used to put the arbiter into a known state before bus operation begins.

Therefore, the required inputs to the arbiter are 'br1', 'br2', 'br3', 'bbsy', and 'reset'. The outputs of the arbiter are 'bg1', 'bg2', 'bg3', and 'bclr'. The initial state machine of the arbiter used as the input to ADACC is shown in Figure 6.11. The only user timing constraints on the arbiter is a default transition time of 240ns.

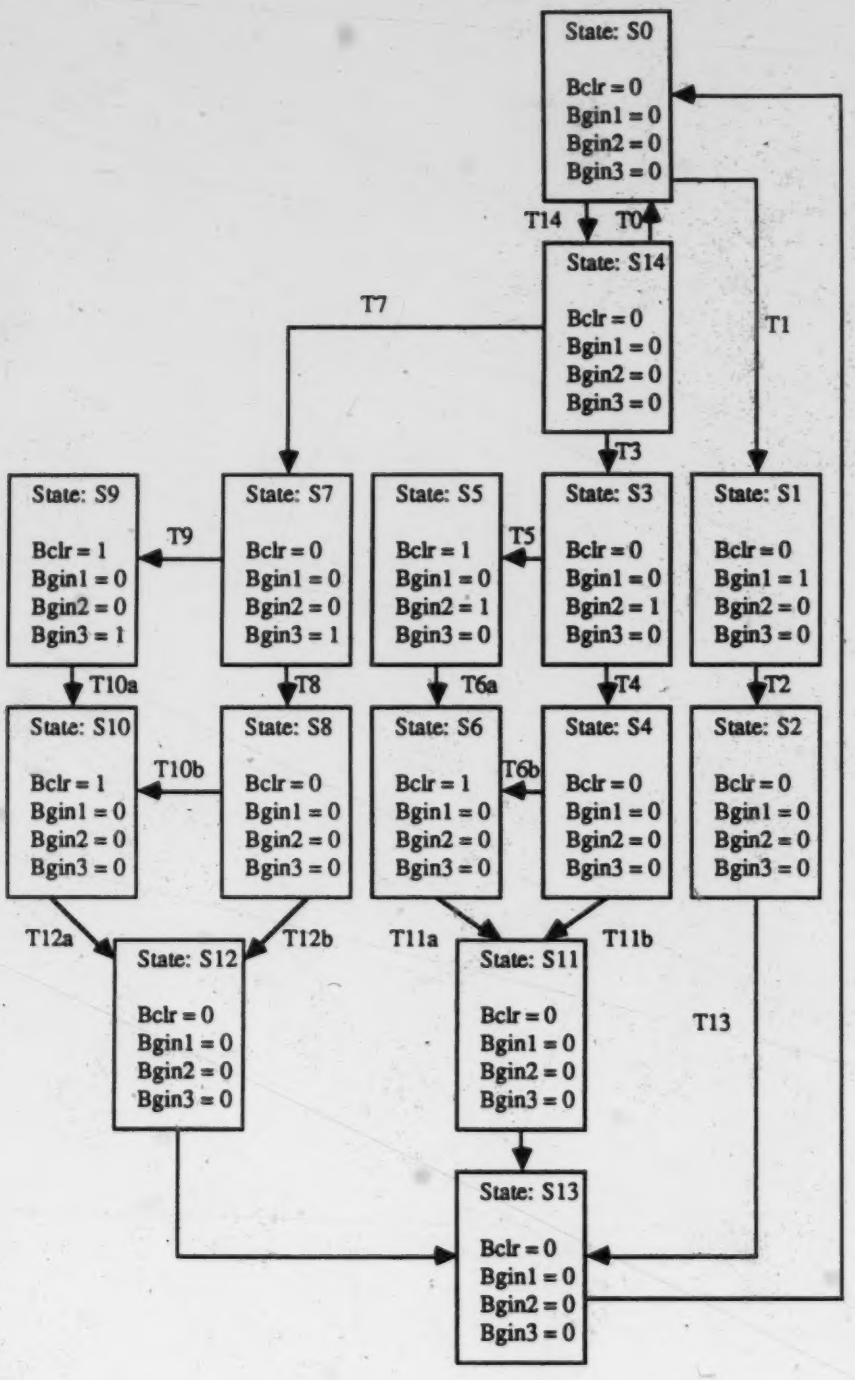

Figure 6.11. Initial VME arbiter state machine

The next state equations are not shown in Figure 6.11 due to page size limitations. Instead, the transitions are marked with identifiers representing the next state equations. Figure 6.12 shows the next state equations for the initial state machine. The textual input to ADACC describing this FSM is included in Appendix E.

Transition Identifier	Next State Equation
T0	reset
T1	br1 * !reset
T2	bbsy + reset
T3	br2 * !reset
T4	!br1 * bbsy + reset
T5	br1 * !reset
T6a	bbsy + reset
T6b	br1 * !reset
T7	!br2 * br3 * !reset
T8	bbsy * !br1 * !br2 + reset
T9	br1 * !reset + br2 * !reset
T10a	bbsy + reset
T10b	br1 * !reset + br2 * !reset
T11a	!bbsy + reset
T11b	!bbsy * !br1 + reset
T12a	!bbsy + reset
T12b	!bbsy * !br1 * !br2 + reset
T13	!bbsy + reset
T14	!br1*br2*!reset + !br1*br3*!reset

Figure 6.12. Next state equations for FSM in Figure 6.11

Automated state assignment required 9 additional transition states, one shared state, and 5 state variables to complete the assignment of the arbiter. The original number of states was 15. One transition state was inserted between state s10 and state s12, three transition states were inserted between state s11 and state s13, three transition states were inserted between state s3 and state s4, and two transition states and one shared state was inserted between state s4 and s6.

After assignment, the logic circuitry for the arbiter was created, the technology mapper was run, and rule-based optimization was performed with topographical rules. Timing optimization reduced the

initial timing violations (totaling 98ns) down to 8ns with the application of 5 topographical rules. However, at this point ADACC could not improve the circuit timing any more, and the system switched to gate delay rules. After the switch to gate delay rules, the 8ns timing violation was eliminated with the application of one rule. The rules used in optimization are listed in Figure 6.13.

1: out = (+, or2_74ls32, (+, or2_74ls32, x, y) , z) → out = (+, or2_74ls32, x, (+, or2_74ls32, z, y))

2: out = (+, or2_74ls32, z, (+, or2_74ls32, y, x)) → out = (+, or2_74ls32, x, (+, or2_74ls32, z, y))

3: out = (*, and2_74ls08, z, (*, and2_74ls08, x, y)) →
out = (*, and2_74ls08, x, (*, and2_74ls08, z, y))

4: out = (*, and2_74ls08, (*, and2_74ls08, x, y) , z) →
out = (*, and2_74ls08, x, (*, and2_74ls08, z, y))

5: out = (*, and2_74ls08, z, (*, and2_74ls08, y, x)) →
out = (*, and2_74ls08, x, (*, and2_74ls08, z, y))

6: out = (+, or2_74ls32, a, b) → out = (+, or2_74hc32, a, b)

Figure 6.13. Rules used in optimization

Rules 1 to 5 are topographical rules and rule 6 is a gate delay rule. Rules 1, 3, 4, 5, and 6 in Figure 6.13 were used once, while rule 2 was used 2 times.

After optimization, the circuit produced by ADACC was tested using SILOS with a default transition time of 240ns. Three different test cases were used. The first case was a general test which allowed the peripherals connected to br1, br2, and br3 to acquire the bus in order. The second case was intended to show the bus request priority in operation, by allowing the peripheral connected to br3 to have the bus first, then having the peripheral connected to br2 force br3 to release the bus, and finally having the peripheral connected to br1 force br2 to release the bus. The third test case was intended to show that if the peripheral connected to br1 had the bus, the peripheral connected to br2 is forced to wait until br1 is done before it is allowed to acquire the bus. The simulation results of these test cases are shown in Figure 6.14, Figure 6.15, and Figure 6.16, respectively.

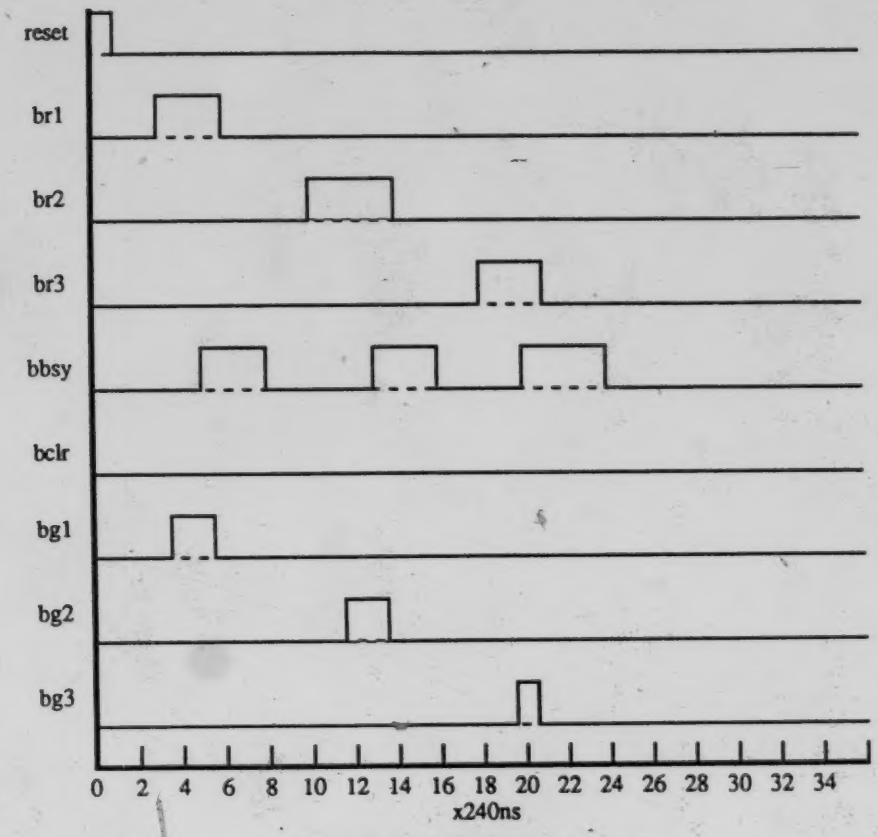

Figure 6.14. VME arbiter test case 1

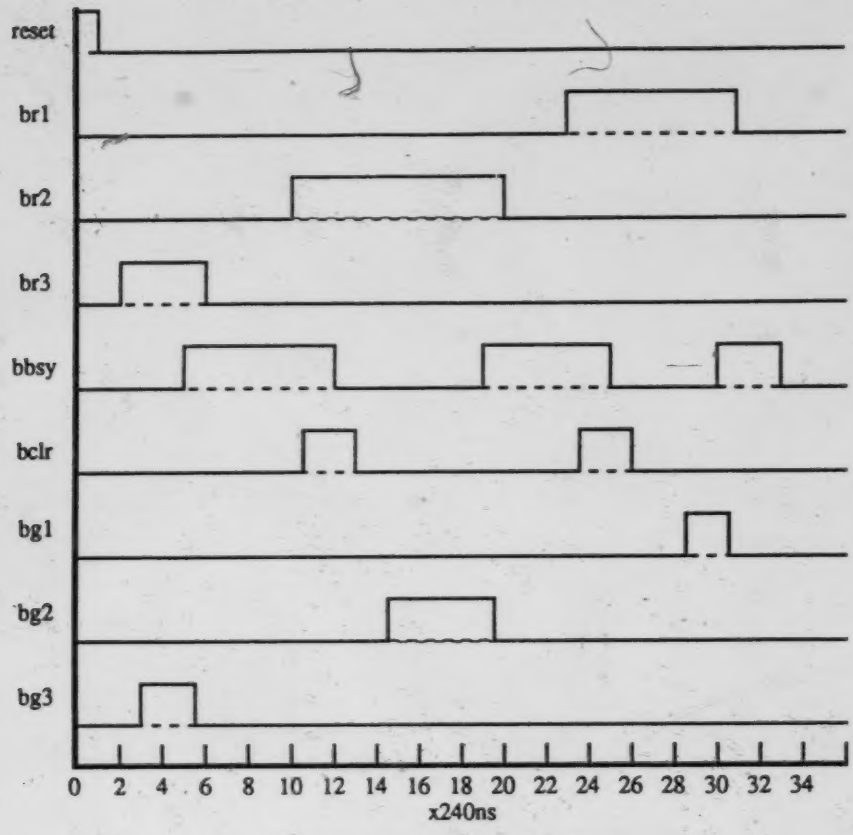

Figure 6.15. VME arbiter test case 2

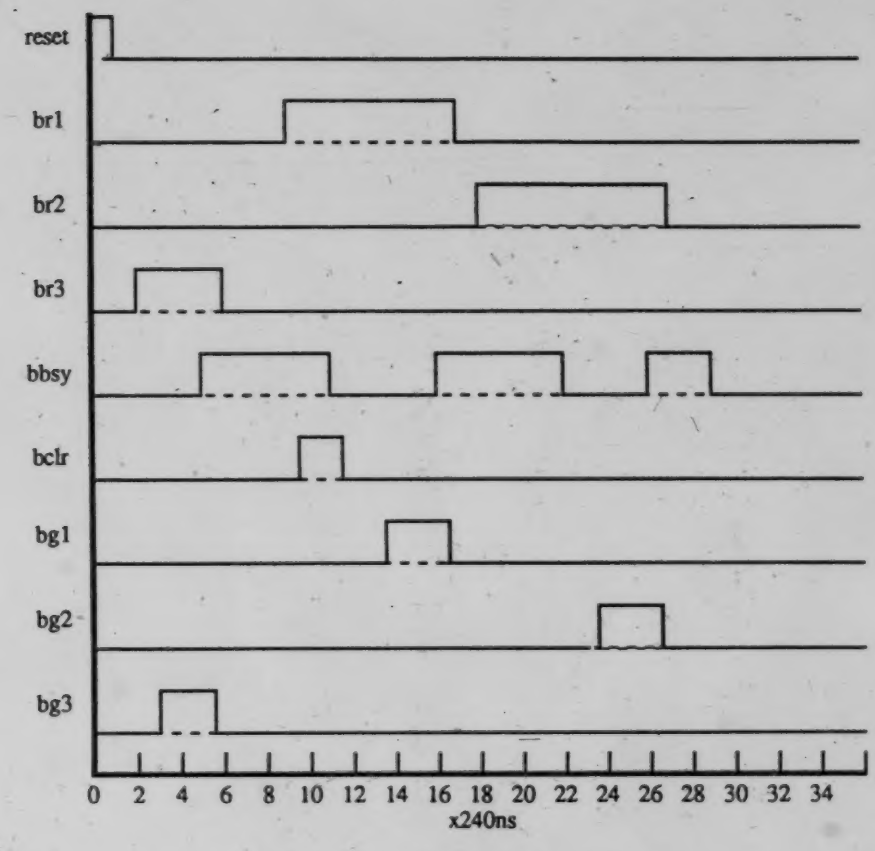

Figure 6.16. VME arbiter test case 3

The automatically generated circuitry passed all three tests without any functional mis-operations or signal glitches, which shows that the circuit did not exhibit any hazards for these test cases. However, there are several places in these tests where it appears that the default transition time restriction has been broken. For example, in test case 1 it takes the arbiter 360ns to assert bg3 after br3 has been asserted. However this delay is due to the three transition stated added between states s11 and s13 that the FSM must pass through after peripheral two releases the bus by lowering 'bbsy'. Each of these transition states takes at most 240ns to traverse, so therefore the extra delay can be expected.

Other examples of apparent speed penalties can be explained in the same manner. In test case 2, the system takes 400ns to assert 'bg2' after the 'bclr' signal is lowered. However, on inspection of the original FSM, it can be seen that the system must traverse through the states s13, s0, and s14 before

'bg2' is asserted, all of which can take a maximum of 240ns to traverse. In test case 3, the system takes 320ns to assert 'bg1' after 'bclr' is lowered, but again on inspection of the FSM, it can be seen that the arbiter must go through the states s13 and s0 before 'bg1' is asserted. Also in test case 3, the arbiter takes 360ns to assert 'bg2' after 'bbsy' is un-asserted. This delay is because the arbiter must again pass through 3 states before 'bg2' is asserted.

6.4. Test Machine 3 - HDLC Protocol Serial Bit Stuffer

HDLC (High-level Data-Link Control) is an ISO layer two protocol used in most modern serial communication systems [Ston83]. This protocol uses a special flag to signal the start and end of a data packet. This flag is 01111110, and is easily recognizable. However, in order to ensure data transparency of the packet information, the flag character *cannot* appear in the data section of the packet. This is achieved by bit stuffing the information in the packet, which is done by inserting a 0 bit after every occurrence of five consecutive 1 bits. Flags are then attached to the packet after bit stuffing is performed.

Here, a circuit has been synthesised to bit-stuff a serial data stream. It is assumed that flags will be added after bit stuffing by a separate circuit. The inputs to the circuit are the input data stream 'd', and a strobe signal 's' used to determine when the input 'd' is valid. Here, 'd' is considered to be valid on a high-to-low transition of the strobe signal. The circuit outputs are the output data stream 'd_out' and a 'hold' signal. This hold signal is used to tell the circuitry generating the original bit stream to hold the input stream for one bit time while a 0 bit is being stuffed into the output data stream. The output 'd_out' is set to be valid on the low-to-high transitions of the strobe signal. That is, the output data stream lags the input data stream by one half bit time.

The original FSM used to implement the stuffing procedure is shown in Figure 6.17.

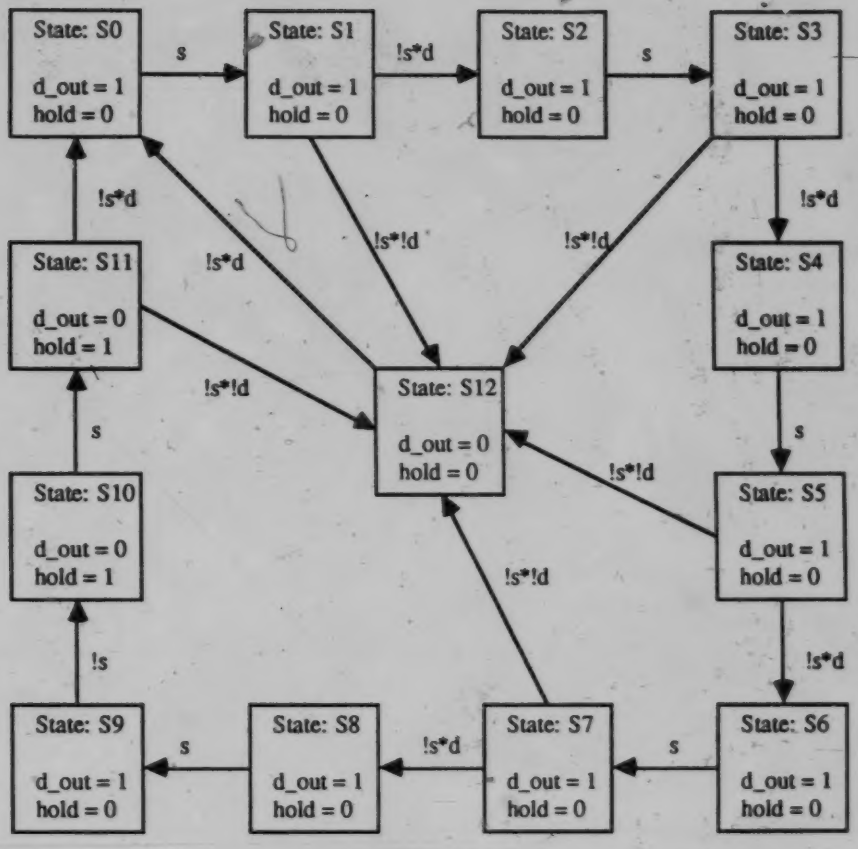

Figure 6.17. Initial FSM for bit stuffer

ADACC assigned the state machine using 5 state variables, 6 transition states, and 1 shared state. Three transition states were added between s12 and s0 and three transition states and one shared state were added between states s7 and s12. The original number of states was 12. This resulted in the assigned FSM shown in Figure 6.18.

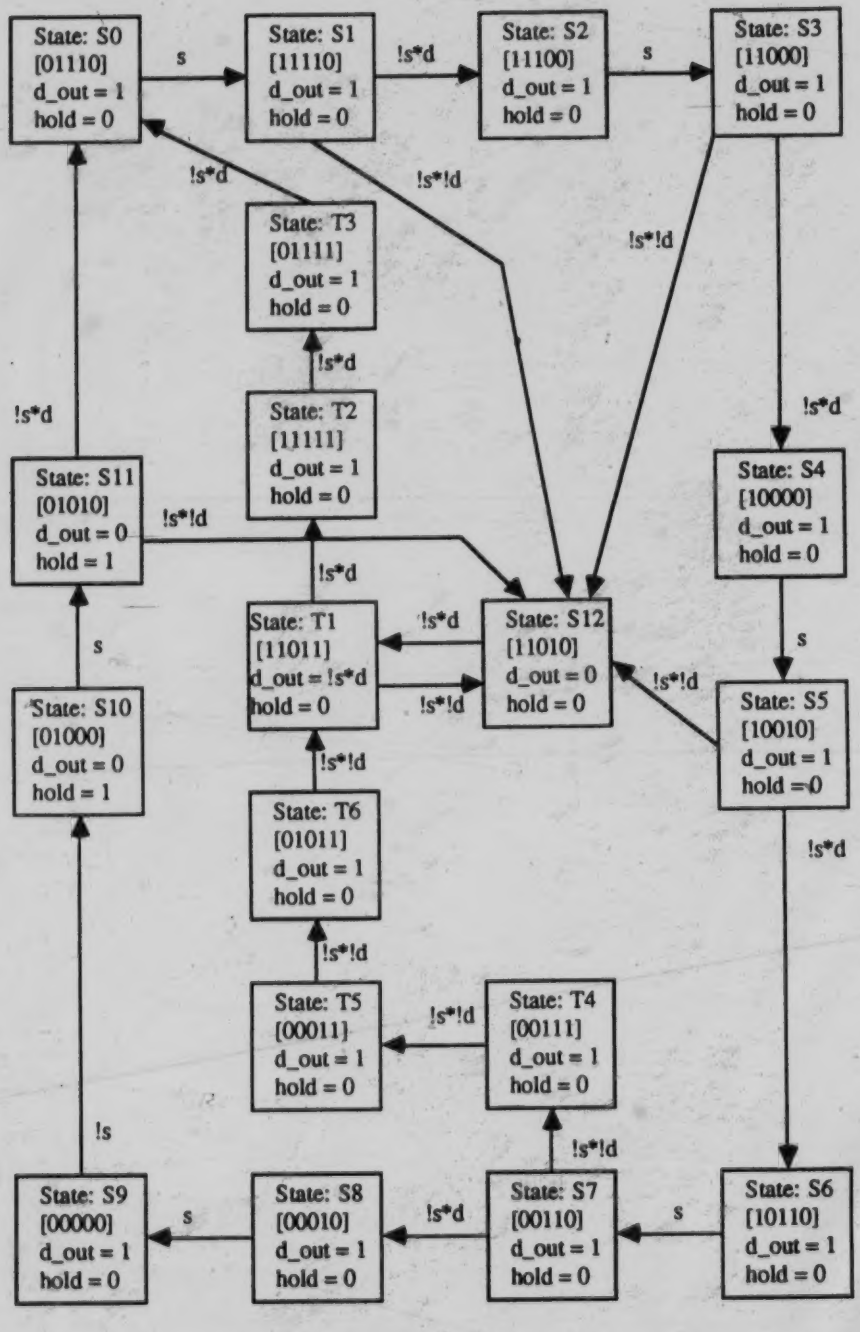

Figure 6.18. FSM for bit stuffer after state assignment

After state assignment, the final circuit was optimized using the topographical rules with a default transition time restriction of 220ns. ADACC managed to reduce an original total timing error of 458ns down to 278ns after the application of the 4 topographical rules listed in Figure 6.19. At this point, ADACC could not find any more rules to apply, so gate delay rules were resorted to to clean up the remaining timing errors.

1: out = (+, or2_74ls32, z, (+, or2_74ls32, x, y)) → out = (+, or2_74ls32, x, (+, or2_74ls32, z, y))

2: out = (+, or2_74ls32, z, (+, or2_74ls32, x, y)) → out = (+, or2_74ls32, y, (+, or2_74ls32, x, z))

3: out = (+, or2_74ls32, (+, or2_74ls32, x, y), z) → out = (+, or2_74ls32, (+, or2_74ls32, z, y), x)

4: out = (+, or2_74ls32, (+, or2_74ls32, x, y), z) → out = (+, or2_74ls32, (+, or2_74ls32, x, z), y)

Figure 6.19. Topographical rules used in optimization

Rule 1 in Figure 6.19 was used 3 times, rule 2 was used once, rule 3 was used 2 times, and rule 4 was used once.

After optimization was completed, the circuit was simulated using the test cases shown in Figures 6.20, 6.21, 6.22, and 6.23. In all of these test cases, the strobe and data inputs are 220ns out of phase in order to avoid breaking the fundamental mode assumption. Note that the time scale used in the test case diagrams has been rounded off to 100ns increments for readability.

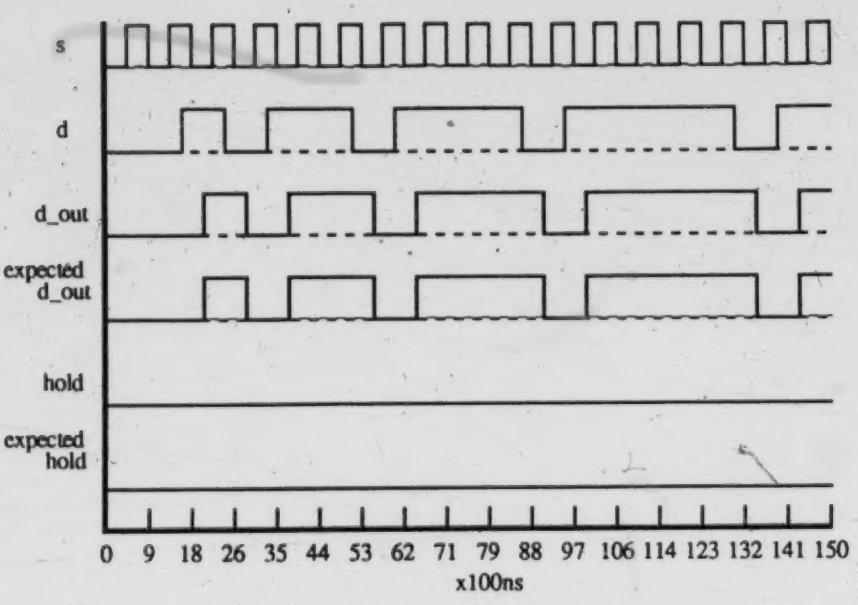

Figure 6.20. Bit stuffer test case 1

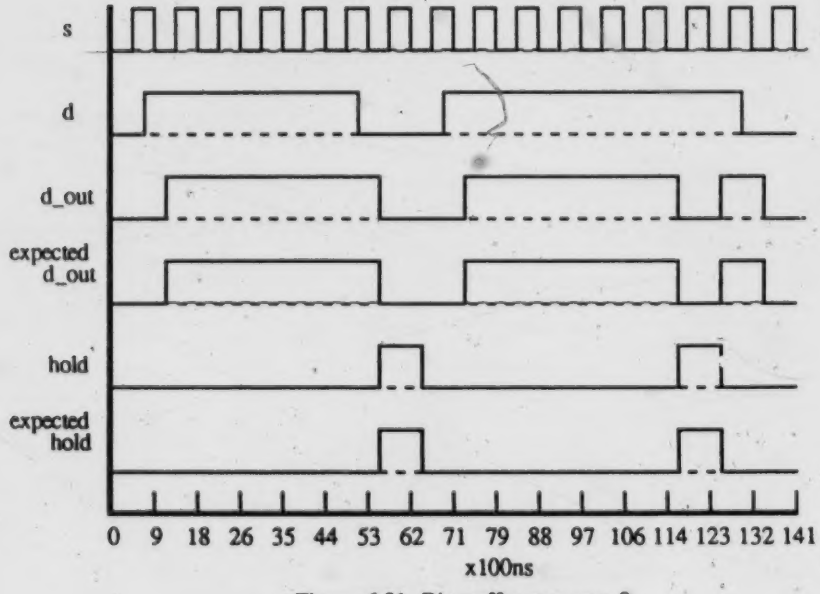

Figure 6.21. Bit stuffer test case 2

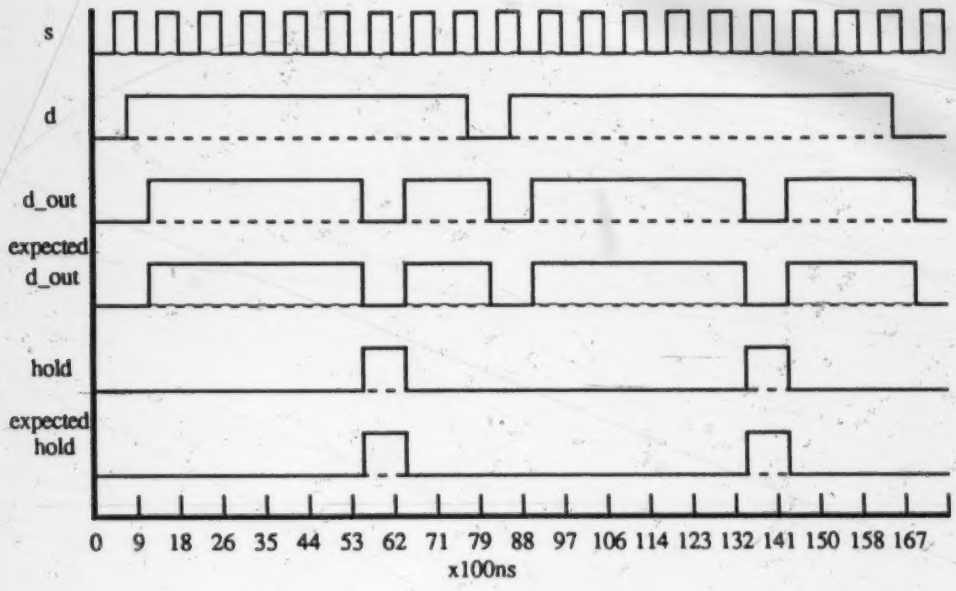

Figure 6.22. Bit stuffer test case 3

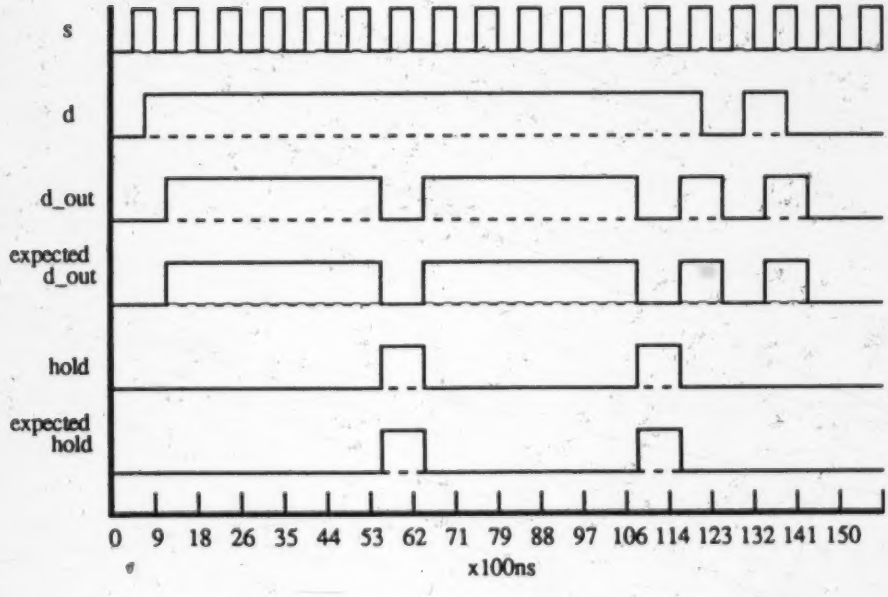

Figure 6.23. Bit stuffer test case 4

The first test in Figure 6.20 shows the operation of the circuit with a simple input data stream that needs no bit stuffing, as the largest number of consecutive ones is only 4. The results of the simulation show that the circuit performs as expected; no bits are stuffed into the output, the output echos the input stream delayed by 1/2 a bit, and the hold output is inactive.

The second test in Figure 6.21 shows the operation of the circuit with an input stream in which there are 5 and 6 consecutive ones respectively. In the first case, a 0 bit is inserted in the output stream, and the hold is activated as soon as 5 ones are detected. The hold signal causes the last bit in the input stream (in this case a 0 bit) to be held for one bit time. This held bit is echoed to the output stream immediately after the stuffed 0 bit. In the second case, the 0 bit is stuffed, and hold is activated after 5 consecutive bits as before, but here the input bit that is held is a 1 bit. This 1 bit is then echoed to the output stream after the stuffed bit.

The third test in Figure 6.22 shows the operation of the circuit with an input stream in which there are 7 and 8 consecutive ones, respectively. Here, 0 bits are stuffed after the 5-th 1 bit in each case, and the remaining bits are echoed to the output.

The fourth test in Figure 6.23 shows the operation of the circuit with an input of 11 consecutive bits. Here, two 0 bits must be stuffed into the output stream, one after each set of 5 consecutive 1 bits.

All of the tests worked without any visible spikes, glitches or other hazards, and the outputs became valid at the most 220ns after the input that caused the change. For example, d_out always changed at most 220ns after the high-to-low transition of the strobe, while in some cases hold changed only 100ns after the high-to-low strobe signal. The delays associated with the transition states are not apparent in the outputs because the behaviour of the outputs in the transition states are set to be the same as the outputs in the state following the transition states. However, each transition state takes at most 220ns to traverse, which does add a speed penalty to the circuit. This penalty was not noticed in the tests because of the generous 880ns period of the strobe signal, which gives plenty of time for traversal of any transition states.

6.5. Test Machine 4 - FM (Single Density) Floppy Disk Data Separator

Data is encoded onto single density floppy disks using a format called the FM, or Manchester code format [Ston83]. In this format, the clock and data are encoded together, each data bit separated with a single clock pulse. The data and clock pulses are 200ns wide, and are separated by a time of 2 microseconds. An example of an FM encoding of a binary bit stream is shown in Figure 6.24.

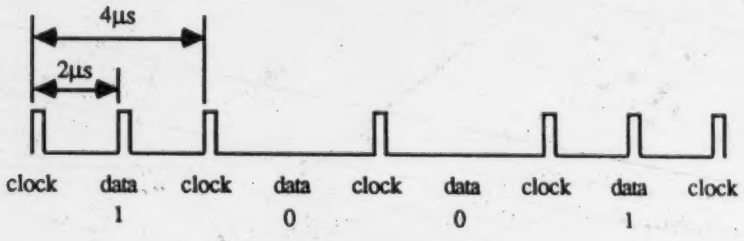

Figure 6.24. FM encoding of a simple bit stream

Writing information to a floppy disk is simply a matter of converting the binary information into a series of data and clock pulses, and then writing this information to the appropriate sectors of the disk.

Reading information from disks is done by using a data separator, which separates the bare pulses from the disk into data and clock information. However, reading the FM encoded data is difficult due to the imperfections of the physical media. The clock and data pulses may be shifted with respect to each other, causing small variations in the timing of the pulses. Data separators incorporate a phase locked loop in order to compensate for the small timing variations of the pulses.

In this test case, a FM data separator is created that incorporates a digital phase locked loop. The data separator takes as input the bit stream from the disk, and a reference clock, and produces an output data stream and an output clock. The output data stream is synchronized to the output clock such that the data can be latched on the falling edge of the clock. The data separator should be able to track the input data with small variations in timing of the input bit stream.

The state machine that implements the data separator is shown in Figure 6.25. Here, the bit stream from disk is called 'di', the reference clock is called '8xc', the output data stream is called 'do', and the output clock is called 'c'. The reference clock used here is eight times as fast as the bit stream from the disk. The state machine is synchronized to state 1 after the first pulse is seen from the disk (which is interpreted to be a clock pulse). Four reference clock periods after this clock pulse, the state machine looks for any data pulse in the input stream, which is then echoed to the output data stream. The state machine then re-synchronizes itself to state 1 on the next clock pulse from the input stream, which takes into account timing fluctuations in the input bit stream.

According to [Ston83], digital FM data separators require a reference clock of at least 16 times the input pulse speed. However, this restriction applies to synchronous FSMs, which can only change states on one edge of the reference clock. This means that only one edge of the reference clock is used to check the timing of the input pulses. In this application, the AFSM can change states on both edges of the reference clock, which allows the machine to check the timing of the pulses from the disk with respect to both edges. Therefore, no timing accuracy is lost by using a reference clock that is eight times the speed of the data pulses instead of 16 times.

The state machine in Figure 6.25 was used as input to ADACC, which produced the state assignment shown in Figure 6.26. This assignment used four state variables, and did not require any shared or transition states.

Rule-based optimization was then performed on the resulting state machine. The only timing restriction used was a default maximum transition time of 125ns. This value was chosen because as the input bit stream has a period of 4 microseconds, the eight times reference clock has a period of 500ns, which means that there is 250ns between every reference clock transition. In order to avoid violating the fundamental mode assumption, the transitions on the input bit stream must not appear during a reference clock transition, which means that a transition can occur once every (250ns / 2) = 125ns.

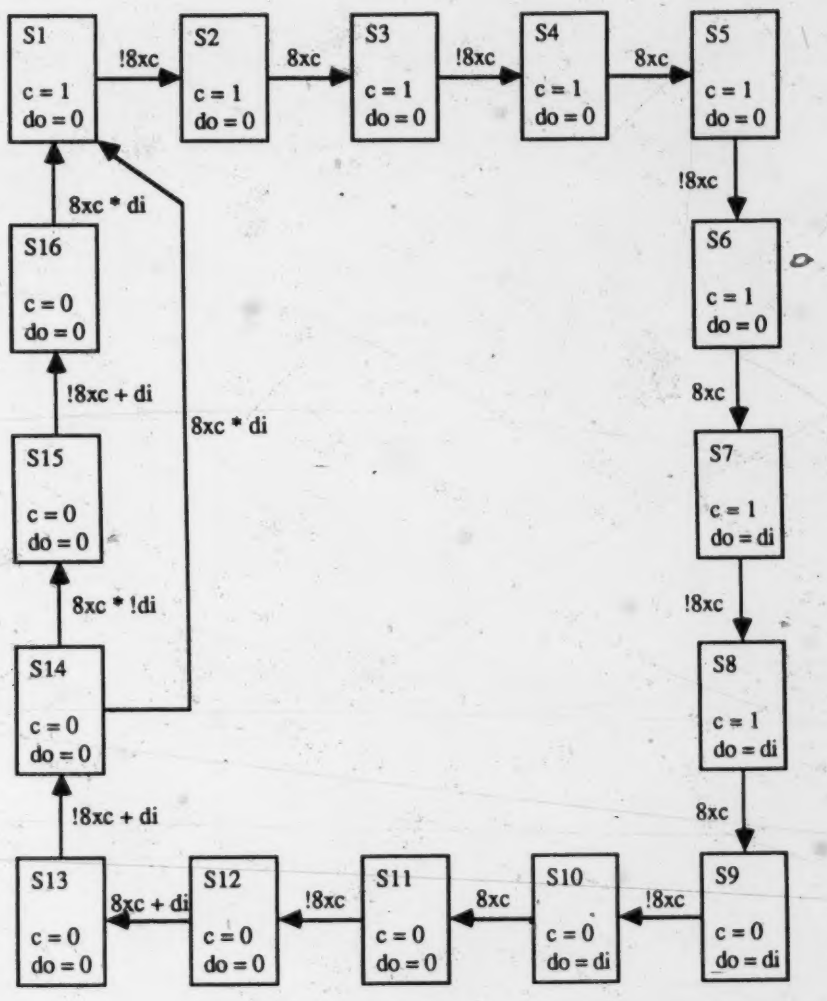

Figure 6.25. Original FSM to implement FM data separator

Unfortunately, ADACC did not remove all of the timing errors in the circuit with the 125ns transition time restriction. Therefore, it was determined that in order to perform testing of the design, the speed of the bit stream from the disk must be slowed down. It was determined that with a default transition time of 150ns, ADACC was able to remove all timing violations in the circuit. This increased the period of the reference clock to 600ns, and the period of the input bit stream to 4.8 microseconds.

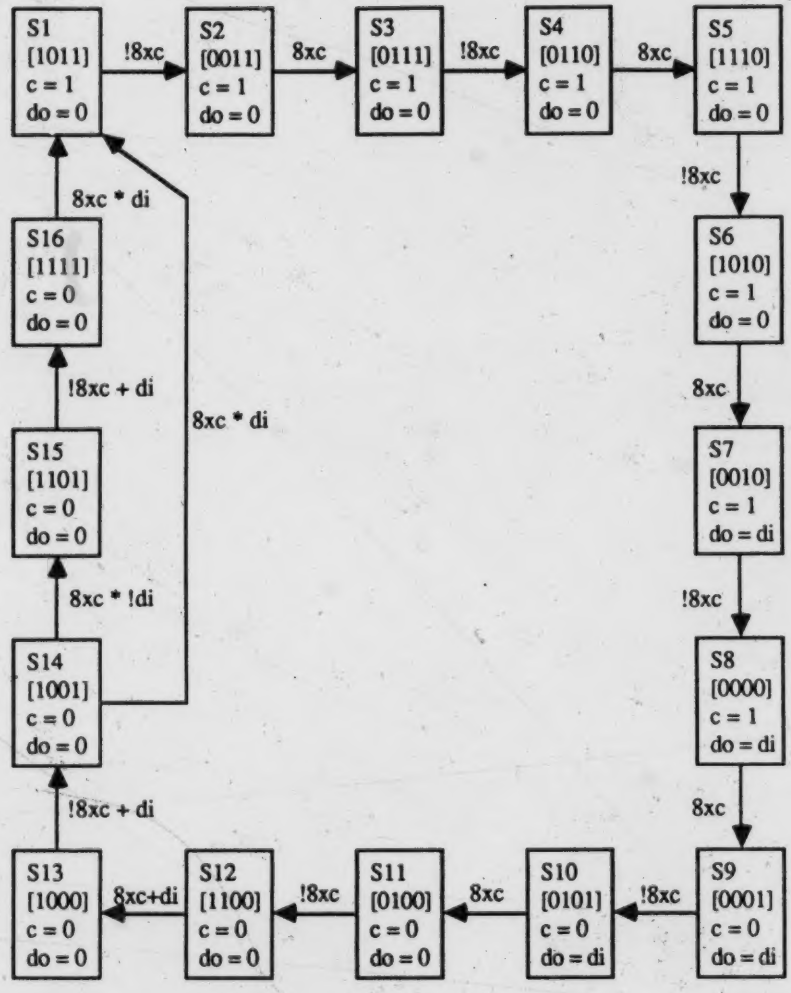

Figure 6.26. FSM to implement FM data separator after state assignment

ADACC's rule-based optimization system used 5 topographical rules to reduce a total timing error of 534ns down to 334ns. At this time, ADACC could not find any more topographical rules to apply, therefore gate delay rules were used to eliminate the remainder of the timing problems so that the circuit could be simulated.

The final circuitry was simulated using SILOS with the three test cases shown in Figure 6.27,

Figure 6.28, and Figure 6.29. The first test case in Figure 6.27 shows the operation of the data separator with an ideal input bit stream that has no bit shifting problems. The second test case in Figure 6.28 shows the operation of the data separator with the input bits shifted forwards and backwards in time by at most 1200ns. The third test case in Figure 6.29 shows data separator operation with a 15% decrease in the period of the input bit stream. Here, the resulting period is equal to 3.6 microseconds.

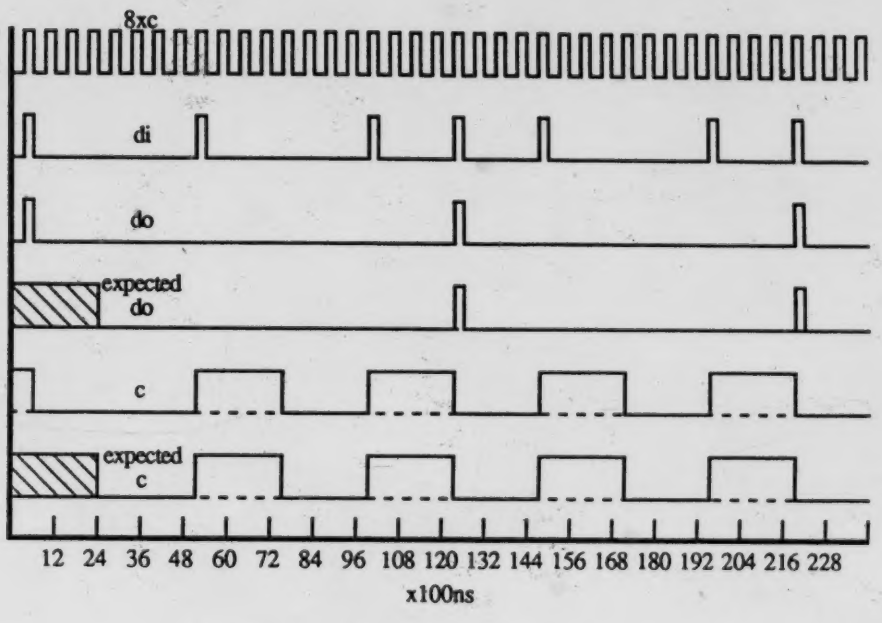

Figure 6.27. FM data separator test case 1

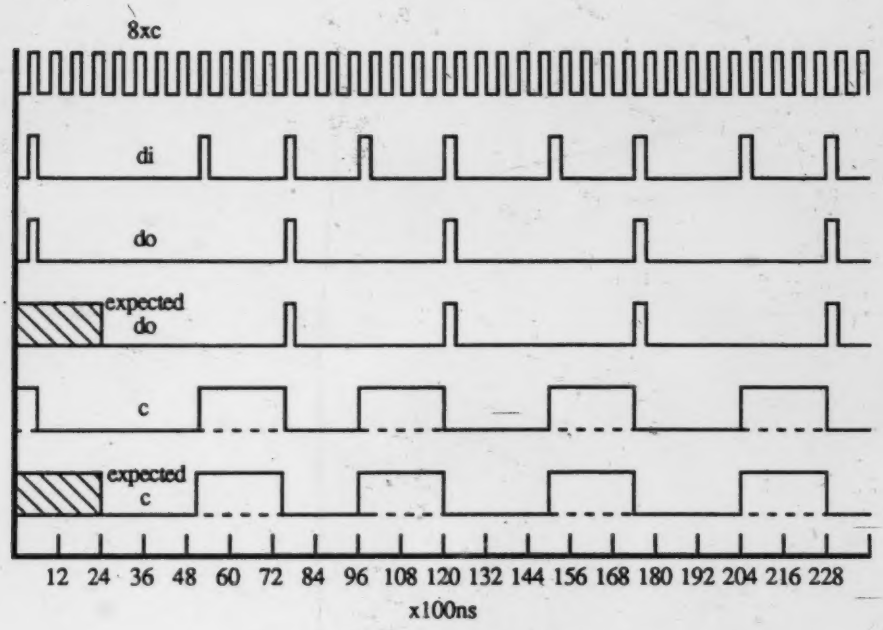

Figure 6.28. FM data separator test case 2

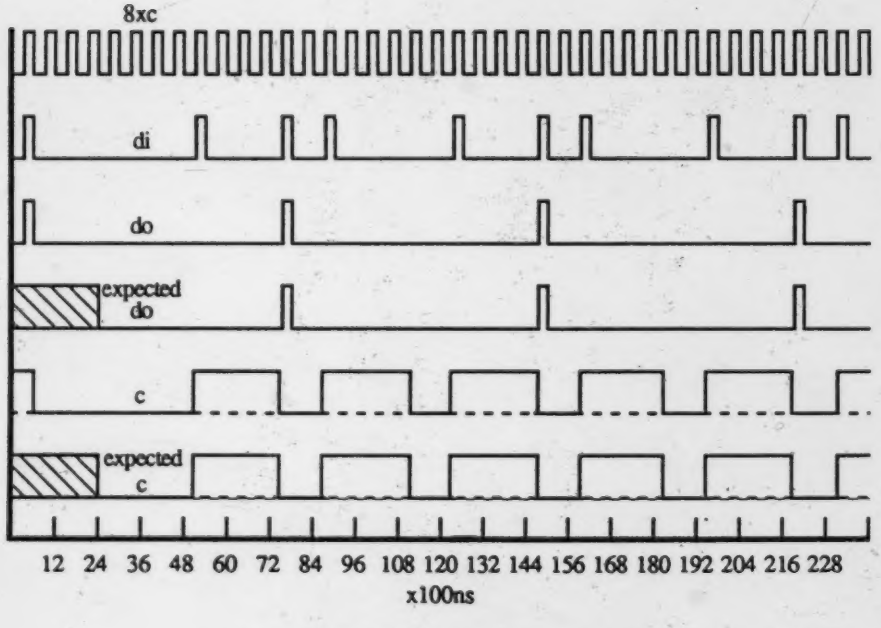

Figure 6.29. FM data separator test case 3

Figure 6.27, 6.28, and 6.29 all show the expected and the actual behaviour of the outputs of the

data separator. Note that the actual behaviour of the clock and data outputs matches the expected behaviour without any glitches or hazards. Also, note that there is a time period at the start of each test case where the digital phase locked loop is synchronizing to the input bit stream. In this time period, the values of the outputs are unknown, as shown by the shaded regions of the timing of the expected behaviour of the circuit outputs.

It is unfortunate that ADACC could not meet the original timing restrictions. However, it should be pointed out that the final circuit performed its function with no flaws when the timing constrains were relaxed. In addition, it is very likely that the required timing would be met with an improved optimization and/or a faster gate technology.

7. Conclusions and Additional Research

In this thesis ADACC, a design tool that synthesizes AFSMs from high-level descriptions has been presented. This tool implements classical FSM design methods using simulated annealing to generate a race-free assignment, and a rule-based timing optimization system to adjust circuit timing to hide essential hazards and to meet most of the user's timing restrictions. In addition to showing that synthesis of these circuits is possible, ADACC demonstrates the usefulness of simulated annealing for state assignment and rule-based timing optimization for elimination of essential hazard effects. ADACC was tested by using it to implement a variety of realistic asynchronous circuits, which were then verified using simulation. In all of these tests, ADACC was capable of generating a correct assignment, correct Boolean equations, and a final circuit netlist that performed the required function without any circuit hazards.

In some test circuits, the required user timing constraints were not met by ADACC with topology modification rules alone: either gate delay rules were required or the user constraints had to be relaxed. It should be stressed that this in no way demonstrates a failure of ADACC. ADACC, or any synthesis system, cannot be expected to meet arbitrary timing constraints because all of the gates in the system have finite delay. However, ADACC depends on timing constraints to eliminate essential hazards as well. The critical difference between these constraints is that the essential hazard constraints limit the speed of particular paths in the circuit with respect to *other paths*, rather than to arbitrary absolute times. Therefore, the success of ADACC in meeting these constraints does not depend on the delays of the logic gates, but instead depends on the power of rule-based optimization to equalize path delays.

In all cases, topographical rules were used to help meet the essential timing constraints, and in some cases these rules eliminated all essential hazard timing problems without resorting to gate delay rules. This shows that the concept of using timing optimization to adjust path delays to help remove the effects of essential hazards is valid, although the power of timing optimization is limited.

7.1. Additional Research

Because of the limited scope of this thesis, there are numerous parts of ADACC that could still be improved. These are discussed in this section.

7.1.1. State Assignment For Reduced Equation Complexity

Currently, the assignment method of ADACC generates race-free, but not optimal assignments, in terms of the complexity of the equations generated using the assignment. At this point, it is not clear whether MTT assignments can be modified to reduce equation complexity, but this is a point that should not be ignored in any future research.

7.1.2. Elimination of Fundamental Mode Assumption

ADACC generates circuitry that must follow the fundamental mode assumption. However, this creates a restriction on the circuit inputs that can limit the usefulness of circuitry produced by ADACC. Therefore, design methods that create AFSMs that do not need to follow the fundamental mode assumption should be researched, and the results of this research included in ADACC.

7.1.3. Optimization Performance

Performance of the rule-based timing optimization portion of ADACC is not up to the standards of commercial logic synthesis systems. This can be seen because of the number of timing errors left by the topographical rule optimization that had to be removed with the use of gate delay rules.

One reason for this is the rather small topographical rule set. This rule set is constrained in that none of the rules can introduce combinational hazards. The rule set is also constrained by the fact that only two input gates are allowed, which removes an entire set of rules that swap between two input gates, and gates with three or more inputs. Nothing can be done about the first constraint, but the second constraint can be improved simply by including more gates in the library and writing the rules to use them.

Another reason for low performance of the timing optimization is that the rule selection method used in ADACC does not allow any uphill moves. This is an important feature of logic synthesis systems like SOCRATES [Geus85], which allows such systems to avoid local minimum solutions. The performance of ADACC's system could be increased with the inclusion of this feature, as SOCRATES did not start producing good results in timing synthesis until uphill moves were allowed.

7.1.4. Optimization for Reduced Area

The rule-based optimization system could also be used to reduce circuit complexity (and therefore area). This use of optimization has been ignored in ADACC in order to keep the implementation as simple as possible. However, it is to any user's advantage to have optimization reduce the size of the resulting circuitry as well as make sure that all timing constraints are met.

7.1.5. Timing Optimization to Increase the Speed of Transition and Shared States

Currently, timing optimization makes every state transition at least as fast as the user-specified default transition time. However, there is no distinction made in the optimizer between states specified by the user, and transition or shared states added by state assignment. Therefore state transitions associated with shared and transition states are made just as fast as user-specified transitions. However, the insertion of transition or shared states increases the number of transitions that the AFSM must go through to traverse one *user-specified* transition. This makes transitions that require transition states appear to the user to be slower than expected. So although state assignment ensures that transition states are not inserted into transitions that must be faster than the default transition time, the transitions that they *are* inserted into are slowed down, which can be considered to be a timing error.

This error can be solved by modifying the default transition time restrictions associated with transitions that include transition states. Here, instead of restricting the delay of *one* transition to be less than or equal to the user-specified default transition time, the total delay of *two or more* transitions should be restricted. Therefore, in future revisions of ADACC default transition time restrictions must

be modified to take into account transition and shared states to ensure that all *user-specified* state transitions meet the default transition time constraint.

7.1.6. Design For Testability

If ADACC is to be used in an IC design environment, testability issues will have to be investigated. The problem is that typical testing methods such as level-sensitive scan design cannot be used with AFSMs because there are no clocked synchronous elements. It is possible to turn an AFSM into a synchronous machine during testing and then change it back after testing is finished, but this method does not test the circuit timing which is required to ensure correct operation.

8. References

[Arms68]
D. B. Armstrong, A. D. Friedman, P. R. Menon, "Realization of Asynchronous Sequential Circuits Without Inserted Delay Elements," IEEE Transactions on Computers, Vol C-17, No. 2, pp. 129-134, February, 1968.

[Arms69]
D. B. Armstrong, A. D. Friedman, P. R. Menon, "Design of Asynchronous Circuits Assuming Unbounded Gate Delays" IEEE Transactions on Computers, Vol C-18, No. 12, pp. 1110-1120, December, 1969.

[Borr87]
G. Borriello and R. H. Katz, "Synthesis and Optimization of Interface Transducer Logic," Proc. 1987 IEEE ICCAD, pp. 274-277.

[Borr88]
G. Borriello, "Combining Event and Data-flow Graphs in Behavioral Synthesis," Proc. 1988 IEEE ICCAD, pp. 56-59.

[Bray87]
R. K. Brayton, R. Rudell, A. L. Sangiovanni-Vincentelli, A. R. Wong, "MIS: A Multiple-Level Logic Optimization System," IEEE Transactions on Computer-Aided Design, Vol. cad-6, No. 6, pp. 1062-1081, 1987.

[Bray84]
R. K. Brayton, G. D. Hachtel, C. T. McMullen, A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis* Boston: Kluwer Academic Publishers, 1984.

[Chu86]
T. Chu, "Synthesis of Self-timed Control Circuits from Graphs: An Example," Proc. 1986 IEEE ICCD, pp. 565-571.

[Darr81]
J. A. Darringer, W. H. Joyner Jr., C. L. Berman, L. Trevillyan, "Logic Synthesis Through Local Transformations," IBM Journal of Research and Development, Vol. 25, No, 4, pp. 272-280, July, 1981.

[Darr84]
J. A. Darringer, D. Brand, J. V. Gerbi, W. H: Joyner Jr., L. Trevillyan, "LSS: A System for Production Logic Synthesis," IBM Journal of Research and Development, Vol. 28, No, 5, pp. 537-545, September, 1984.

[Geus85]
A. J. de Geus, W. Cohen, "A Rule-Based System for Optimizing Combinational Logic," IEEE Design and Test, pp. 22-32, August, 1985.

[Hack71]
R. O Hackbart, D. L. Dietmeyer, "The Avoidance and Elimination of Function Hazards in Asynchronous Sequential Circuits," IEEE Transactions on Computers, Vol C-20, No. 2, pp. 184-189, February, 1971.

[Hlav70]
J. Hlavicka, "Essential Hazard Correction Without the Use of Delay Elements," IEEE Transactions on Computers, Vol C-19, No. 2, pp. 232-238, 1970.

[Kirk83]
S. Kirkpatrick, C. D. Gelatt Jr., M. P. Vecchi, "Optimization by Simulated Annealing," Science, Vol 220, No. 4598, pp. 671-680, May 1983.

[Kuhl78]
J. G. Kuhl, S. M. Reddy, "A Multicode Single Transition Time State Assignment for Asynchronous Sequential Machines," IEEE Transactions on Computers, Vol. C-27, No. 10, pp. 927-934, October, 1978.

[Lang69]
O. G. Langdon, "Delay-Free Asynchronous Circuits with Constrained Line Delays," IEEE Transactions on Computers, Vol C-18, No. 2, pp. 175-181, February, 1969.

[Lea89]
Doug Lea, *User's Guide to GNU C++ Library* Free Software Foundation, 1989.

[Lewi71]
D. W. Lewin, "Advanced Aspects of Asynchronous Logic Design," Computer Journal, Vol 14, No. 1, pp. 254-259, 1971.

[Lewe74]
Douglas Lewin, *Logical Design of Switching Circuits, 2nd ed.* London: Thomas Nelson and Sons Ltd., 1974.

[MacL83]
B. J. MacLennan, *Principles of Programming Languages* New York: Holt, Rinehart and Winston, 1983.

[May90-1]
T. C. May, E. F. Girczyc, "State Assignment For Asynchronous State Machines," To Appear in Proc. of IEEE 33rd. Synp. on Circuits and Systems, 1990.

[May90-2]
T. C. May, E. F. Girczyc, "Simulated Annealing with Algorithmic Cleanup for MTT Assignments of Asynchronous State Machines," Proc. CCVLSI '90, 1990.

[Mccl65]
E. J. McCluskey, *Introduction To The Theory of Switching Circuits* New York: McGraw-Hill Book Company, 1965.

[Mccl86]
E. J. McCluskey, *Logic Design Principles With Emphasis on Testable Semicustom Circuits*. New Jersey: Prentice-Hall, 1986.

[Mead80]
C. Mead, L. Conway, *Introduction to VLSI Systems*, California: Addison-Wesley Co., 1980.

[Meng89]
T. H. Y. Meng, R. W. Brodersen, D. G. Messerschmitt, "Automatic Synthesis of Asynchronous Circuits from High-Level Specifications," IEEE Transactions on CAD, Vol. 8, No. 11, Nov. 1989.

[Mill65]
R. E. Miller, *Switching Theory, Vol. 2: Sequential Circuits and Machines*, New York: John Wiley & Sons, 1965.

[Moto85]
Motorola, *VMEbus Specification Manual*. Arizona: Micrology pbt, 1985.

[Nany79]
T. Nanya, Y. Tohma, "Universal Multicode STT State Assignments for Asynchronous Sequential Machines," IEEE Transactions on Computers, Vol. C-28, No. 11, pp. 811-818, November, 1979.

[Rich83]
E. Rich "Artificial Intelligence," New York: McGraw-Hill, 1983.

[Roth79]
C. H. Roth, *Fundamentals of Logic Design* Second Edition, West Publishing Company, 1979.

[Sarm88]
D. Sarma and L. N. Kannan, "Automated Synthesis of Finite State Machines," VLSI Technical Bulletin, Vol 3, No. 3, pp. 48-55, 1988.

[Sech85]
C. Sechen, A. Sangiovanni-Vincentelli, "The TimberWolf Placement and Routing Package," IEEE Journal of Solid-State Circuits, Vol SC-20, No. 2, pp. 510-522, April 1985.

[Ston83]
H. S. Stone, *Microcomputer Interfacing*. Reading, Massachusetts: Addison-Wesley Publishing Co., 1983.

[Stro86]
Bjarne Stroustrup, *The C++ Programming Language* Ontario: Addison-Wesley Publishing Company, 1986.

[Unge59]
S. H. Unger, "Hazards and Delays in Asynchronous Sequential Switching Circuits," IRE Transactions on Circuit Theory, Vol. CT-6, no. 1, pp. 12-25, March, 1959.

[Unge69]
S. H. Unger, *Asynchronous Sequential Switching Circuits*, New York: Wiley-Interscience, 1969.

[Whit84]
S. R. White, "Concepts of Scale in Simulated Annealing," IEEE ICCD, pp. 646-651, 1984.

[Wins84]
P. H. Winston, and B. K. P. Horn, *Lisp* Second Edition, Massachusetts: Addison-Wesley Publishing Company, 1984.

[Ziss72]
D. Zissos, *Logic Design Algorithms*. London: Oxford University Press, 1972.

[Ziss79]
D. Zissos, *Problems and Solutions in Logic Design*. Oxford: Oxford University Press, 1979.

9. Appendix A, Syntax of ADACC Parsers

9.1. Finite State Machine Specification Syntax

This appendix describes the syntax used to describe the user's finite state machine. The following can be specified in this syntax:

1. the states in the machine
2. the behaviour of the outputs in each state. Both Mealy and Moore outputs are allowed.
3. the state transitions between states
4. transition next state equations
5. the user timing restrictions.

This syntax is free-format, which means that keywords can be placed at any position of a line and still be acceptable. There must be at least one 'white space' (blank or tab) character between each keyword or identifier.

In most state machine specification languages, state transitions are specified as part of the states. For example, associated with each state would be several "next state" fields. Each one of these fields would contain a next state equation, and the name of the state the machine would enter if the next state equation is true. However, in order to keep this parser simple, transitions are specified independently of states. This greatly helps in reducing complexity without causing a great burden to the user.

9.1.1. Overall Syntax

BNF [MacL83] is used to described the input syntax. The syntax to fully describe an input state machine is presented below:

```
<state_machine_input> ::=
        [<default_tt>]
        <input>+
        <state>+
        <arc>+
        <end>
```

9.1.2. State Syntax

The state syntax is defined in the following BNF expression:

```
<state> ::= STATE:
        <name_field>
        <output_field>+
        [<assignment_field>]
        END : STATE;
```

9.1.2.1. Name Field

This field is intended to allow the user to name the state. The name field is specified as follows:

```
<name_field> ::= NAME : <state_name> ;
<state_name> ::= <ASCII_string>
```

No two states are allowed to have the same name.

9.1.2.2. Output Field

This field allows the user to specify the behaviour of the outputs in a particular state. This field is specified as follows:

```
<output_field> ::= OUTPUT: <output_expression> ;
<output_expression> ::= <output_name> = <boolean_expression>
<output_name> ::= <ASCII_string>
```

There must be consistency in the output fields of every state in the machine. That is, each state must have the same number of output fields, and the output names of each of these fields must be identical from state to state.

9.1.2.3. Boolean Expressions

These expressions represent Boolean equations. The BNF description for these expressions follows:

```
<boolean_expression> ::= <product_term>
                       | <boolean_expression> <OR_char> <product_term>
<product_term> ::= <variable>
                       | <product_term> <AND_char> <variable>
<variable> ::= [<NOT_char>] <ASCII_string>

<NOT_char> ::= !
<AND_char> ::= & | *
<OR_char> ::= # | +
```

The variables used in these expressions must be declared at the top of the file as input variables. Declaring input variables will be discussed in a following section.

9.1.2.4. Assignment Field

This field allows the user to specify a state assignment for each state. This field is specified as follows:

```
<assignment_field> ::= ASSIGN: <bit_vector> ;
<bit_vector> ::= <bit_char>
               | <bit_vector> <bit_char>
<bit_char> ::= 0 | 1
```

9.1.2.5. Examples of State Descriptors

The following are examples of valid state descriptors:

```
STATE :
    NAME : s1 ;
    OUTPUT : op1 = !a * b * !d;
    OUTPUT : op2 = clock # data ;
    ASSIGN : 0010;
END : STATE;

STATE : NAME : s2; OUTPUT : op1 = b # c; OUTPUT : op2 = 0; END : STATE;
```

9.1.3. Transition Arc Syntax

Each transition arc must have a minimum of three fields associated with it: a FROM field, a TO field, and an ON field. The FROM field describes the previous state of the arc, and the TO field describes the next state of the arc. That is, the state machine will move from the state described in the FROM field to the state described in the TO field if this arc is traversed. The ON field holds the next state equation that must be true if this arc is to be traversed. An additional field, called the CRITICAL field, can be used to specify timing constraints on the state traversal. This field is optional. The basic arc syntax is as follows:

```
<arc> ::= ARC :
    [<critical_field>]
    <from_field>
    <to_field>
    <on_field>
    END : ARC;
```

9.1.3.1. Critical Field

As stated previously this field is used to specify timing constraints. The syntax for this field is as follows:

```
<critical_field> ::= CRITICAL : <max_time> ;
```

where <max_time> is an integer that specifies the maximum time this transition can take and still be acceptable. It is specified in nanoseconds.

9.1.3.2. From Field

This field specifies the state that this arc comes from. Its syntax is as follows:

```
<from_field> ::= FROM : <state_name> ;
```

where <state_name> is the state the transition is leading to.

9.1.3.3. To Field

This field specifies the state that this arc goes to. Its syntax is as follows:

```
<to_field> ::= TO : <state_name> ;
```

where <state_name> is the state the transition is comming from.

9.1.3.4. On Field

This field describes the conditions that need to be true if this arc of the state machine is to be traversed. Its syntax is as follows:

<on_field> ::= ON : <boolean_expression> ;

9.1.3.5. Arc Examples

The following are examples of correct arc syntax:

```
ARC :
      FROM : s1 ;
      TO :   s2 ;
      ON :   a * b ;
END : ARC;

ARC :
      CRITICAL : 100000 ;
      FROM : s2 ;
      TO   : s3 ;
      ON   : clock * !c ;
END : ARC ;
```

9.1.4. Input Variables

The purpose of these expressions is to define the names of the inputs of the state machine. These input variables must be declared before any Boolean expression. Therefore, these declarations must be first in the file.

The syntax of these declarations is as follows:

<input> ::= INPUT : <input_variable> ;
<input_variable> ::= <ASCII_string>

Example input declarations follow:

```
INPUT : clock ;
INPUT : data ;
INPUT : a ;
```

9.1.5. Additional Syntax

The user can also express the default maximum transition time that the final circuit must follow. This time is specified with the following phrase:

<default_tt> ::= DEFAULT_TT : <max_time> ;

Where <max_time> is the default transition time time specified in nanoseconds. The default transition time can be specified anywhere in the input file, but it can only be specified once. The default value is 100ns.

In addition to the above, the state machine source file must end with the following phrase:

<end> ::= END : PARSE ;

9.1.6. Input File Example

The following is a complete example of an actual state machine input file:

```
INPUT: ip1 ;
INPUT: ip2;
INPUT: ip3     ;
INPUT: a;
INPUT: b;
INPUT: c;
INPUT: d;
INPUT: clock;
INPUT: data;


STATE : NAME : s1; OUTPUT : op1 = !a; OUTPUT : op2 = b; ASSIGN : 0010;
END : STATE;

STATE : NAME : s2; OUTPUT : op1 = c; OUTPUT : op2 = d; ASSIGN: 0000;
      END : STATE;

STATE : NAME : s3; OUTPUT : op1 = !a* b *!d; OUTPUT : op2 = clock * data;
      ASSIGN : 0011; END : STATE;

ARC : CRITICAL : 100; FROM : s1; TO : s3; ON : ip1 * !ip2; END : ARC;

ARC :
      CRITICAL : 300;
      FROM : s2;
      TO : s1;
      ON : ip2 # ip3;
END : ARC;

ARC :
      FROM : s3;
      TO : s2;
      ON : !ip3;
END : ARC;

END : PARSE;
```

9.1.7. AFSM Parser Reserved Words

The following words are reserved and should not be used in any <ASCII_string>:

1 ARC
2 ASSIGN
3 CRITICAL
4 END

5 FROM
6 INPUT
7 NAME
8 OUTPUT
9 ON
10 PARSE
11 QN (where 'N' is any integer)
12 STATE
13 TO
14 N-B_split (where 'N' is any integer)
15 N-T_Split (where 'N' is any integer)
16 N-Trans (where 'N' is any integer)
17 DEFAULT_TT

9.2. Circuit Parser Syntax

Circuits in ADACC are specified by a list of gate descriptor strings, one string for each output of the digital circuit. Each circuit is specified as follows:

<circuit> ::= <output_name> = <gate_expression>
<output_name> ::= <ASCII_string>
<gate_expression> ::= (<op>, <gate_name>, <input_expressions>)
| (<unary_op>, <gate_name>, <input_expression>)
<op> ::= <AND_char> | <OR_char>
<unary_op> ::= <NOT_char>
<input_expressions> ::= <input_expressions> , <input_expression>
<input_expression> ::= <input_name> | <gate_expression>
<input_name> ::= <ASCII_string>

Some examples of circuit descriptions are:

d = (+, or3, a, b, c)
out = (+, or2_74ls32, (*, and2_74ls08, a, b), c)
output = (!, inv, (*, and4, a, b, c, d))

9.3. Optimization Rule Syntax

Modus ponens rules for optimization are specified as two different circuits that implement the same function. A typical rule is specified as follows:

<rule> ::= <LHS_circuit> → <RHS_circuit>
<LHS_circuit> ::= <circuit>
<RHS_circuit> ::= <circuit>

The term 'LHS_circuit' represents the circuit on the LHS of the rule, and the term 'RHS_circuit' represents the circuit on the RHS of the rule. Both of these circuits are specified using the circuit syntax described in the previous section.

Examples of typical rules are as follows:

out1 = (+, or4, a, b, c, d) → out1 = (+, or2, (+, or2, a, b), (+, or2, c, d))
out2 = (!, inv, (!, inv, (!, inv, (!, inv, in1)))) → out2 = in1

10. Appendix B, Optimization Gate Library

This appendix contains descriptions of all of the gates in the technology file used in the optimization section of ADACC. These gates are based on TTL 7400 series chips. Figure B.1 contains all of the inverter gates, Figure B.2 contains all of the AND gates, and Figure B.3 contains all of the OR gates used.

Gate Name	Gate Speed
inv_74as04	2 ns
inv_74hc04	5 ns
inv_74ls04	9 ns

Figure B.1. Inverters in Optimization Gate Library

Gate Name	Gate Speed
and2_74as08	3 ns
and2_74hc08	6 ns
and2_74ls08	12 ns

Figure B.2. AND gates in Optimization Gate Library

Gate Name	Gate Speed
or2_74as32	4 ns
or2_74hc32	9 ns
or2_74ls32	17 ns

Figure B.3. OR gates in Optimization Gate Library

11. Appendix C, Important methods of the data and graph class

11.1. Common Methods of The Data Class

All of the children of the data class share a common set of basic methods. These methods are as follows:

```
int data::equal(data*) /* returns true if the two objects are equal */
int data::eq(data*) /* returns true of the two objects are the same */
data *data:create_copy() /* returns a pointer to a copy of this object */
int data::parse(istream*) /*parses ASCII input that represents the object*/
int data::un_parse(ostream*) /*creates ASCII representation of info in object*/
```

The 'equal()' method returns a true value if the passed data object contains the same information as the calling object. The 'eq()' method returns true if the passed data object is the *same* as the calling object. The 'create_copy()' method creates a copy of the calling data object, returning a pointer to the copy.

The 'parse()' and 'un_parse()' methods are responsible for reading and writing ASCII representations of the information in the object. This allows relatively simple implementations of the various parsers that are used in ADACC.

11.2. Graph Class Traversal Methods

The graph class supports both sequential and topographical traversal methods. The sequential methods are described below:

1. data *graph::first_node() /* returns a pointer to the data field of the first node */
2. data *graph::next_node(data*) /*returns a pointer to the data field of the next node*/
3. data *graph::first_arc() /* returns a pointer to the data field of the first arc */
4. data *graph::next_arc(data*) /*returns a pointer to the data field of the next arc*/

The method 'first_node()' returns a pointer to the data object that is associated with the first node that was added to the graph. The method 'next_node(data*)', when passed a pointer to a data object in the graph, returns the data object that is associated with the node that was added after the node associated with the passed data object. The methods 'first_arc()' and 'next_arc(data*)' operate in the same way, with the exception that data objects that are associated with the arcs are returned.

Topographical traversal is used when the topology of the graph needs to be looked at. Here, the graph can be traversed from any node to any arc that it is attached to, and then these arcs can be further traversed to any node they attach to. The following methods implement topographical traversal:

1. data *graph::a_next_node(data*)
2. data *graph::a_prev_node(data*)
3. data *graph::first_next_arc(data*)
4. data *graph::next_next_arc(data*, data*)
5. data *graph::first_prev_arc(data*)
6. data *graph::next_prev_arc(data*, data*)

The first and second methods allow traversal from an arc to either the next node connected to that arc (if the first method is used), or the previous node connected to that arc (if the second method is used). A pointer to the data object associated with the arc to be used is passed to them both, and a pointer to the data object associated with the previous or next node is returned.

The methods 3, 4, 5, and 6 allow traversal from a node to any arc that it is attached to. These arcs are divided into two sub-classes: 'previous' arcs (also known as incomming arcs), and 'next' arcs (known as outgoing arcs). The method 'first_prev_arc()' returns the data pointer of the first 'prev' arc, if it is passed the data pointer of the starting node. The method 'next_prev_arc()' then returns the data pointer of the next 'prev' arc, if it is passed the data pointer of the starting node, and the data pointer of the first 'prev' arc. If there is no next arc, then zero is returned. The methods 'first_next_arc()', and 'next_next_arc()' operate in a similar fashion, with the exception that they return pointers to the data objects associated with the next arcs.

12. Appendix D, Technology Mapping and Timing Optimization Rules

This appendix contains all of the technology mapping and timing optimization rules used in ADACC. Please see Appendix A for a description of the rule syntax.

12.1. Technology Mapping Rules

The technology mapping rules for inverters are shown in Figure D.1, the technology mapping rules for AND gates are shown in Figure D.2, and the technology mapping rules for OR gates are shown in Figure D.3.

```
out = (!, inv, x) → out = (!, inv_74ls04, x)
```

Fig D.1 Inverter Technology Mapping Rules

```
out = (*, and2, x, y) → out = (*, and2_74ls08, x,y)

out = (*, and3, x, y, z) → out = (*, and2_74ls08, x,(*, and2_74ls08, y, z))

out = (*, and4, a, b, c, d) → out = (*, and2_74ls08, (*, and2_74ls08, a, b),(*, and2_74ls08, c, d))

out = (*, and5, x, y, z, x2, y2) → out = (*, and2_74ls08, (*, and2_74ls08, x,y), (*, and3, z, x2, y2))

out = (*, and6, x, y, z, x2, y2, z2) → out = (*, and2_74ls08, (*, and3, x,y,z),(*, and3,x2,y2,z2))

out = (*, and7,a,b,c,d,e,f,g) → out = (*, and2_74ls08, (*, and3, a,b,c),(*, and4 ,d,e,f,g))

out = (*, and8,a,b,c,d,e,f,g,h) → out = (*, and2_74ls08, (*, and4, a,b,c,d),(*, and4 ,e,f,g,h))

out = (*, and9,a,b,c,d,e,f,g,h,i) → out = (*, and2_74ls08, (*, and4, a,b,c,d),(*, and5 ,e,f,g,h,i))

out = (*, and10,a,b,c,d,e,f,g,h,i,j) → out = (*, and2_74ls08, (*, and5, a,b,c,d,e),(*, and5 ,f,g,h,i,j))

out = (*, and11,a,b,c,d,e,f,g,h,i,j,k) → out = (*, and2_74ls08, (*, and5, a,b,c,d,e),(*, and6 ,f,g,h,i,j,k))

out = (*, and12,a,b,c,d,e,f,g,h,i,j,k,l) → out = (*,and2_74ls08,(*,and6, a,b,c,d,e,f),(*, and6 ,g,h,i,j,k,l))

out = (*, and13,a,b,c,d,e,f,g,h,i,j,k,l,m) → out = (*,and2_74ls08,(*,and6,a,b,c,d,e,f),(*,and7 ,g,h,i,j,k,l,m))

out = (*, and14,a,b,c,d,e,f,g,h,i,j,k,l,m,n) → out = (*,and2_74ls08,(*,and7,a,b,c,d,e,f,g),(*,and7,h,i,j,k,l,m,n))

out = (*, and15,a,b,c,d,e,f,g,h,i,j,k,l,m,n,o) → out=(*,and2_74ls08,(*,and7,a,b,c,d,e,f,g),(*,and8,h,i,j,k,l,m,n,o))

out = (*, and16,a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p) → out=(*,and2_74ls08,(*,and8,a,b,c,d,e,f,g,h),(*,and8,i,j,k,l,m,n,o,p))
```

Fig. D.2. AND Gate Technology Mapping Rules

```
out = (+, or2, x, y) → out = (+, or2_74ls32, x,y)

out = (+, or3, x, y, z) → out = (+, or2_74ls32, x,(+, or2_74ls32, y, z))

out = (+, or4, x, y, z, z2) → out = (+, or2_74ls32, (+, or2_74ls32, x, y),(+, or2_74ls32, z, z2))

out = (+, or5, x, y, z, x2, y2) → out = (+, or2_74ls32, (+, or2_74ls32, x,y), (+, or3, z, x2, y2))

out = (+, or6, x, y, z, x2, y2, z2) → out = (+, or2_74ls32, (+, or3, x,y,z),(+, or3,x2,y2,z2))

out = (+, or7,a,b,c,d,e,f,g) → out = (+, or2_74ls32, (+, or3, a,b,c),(+, or4 ,d,e,f,g))

out = (+, or8,a,b,c,d,e,f,g,h) → out = (+, or2_74ls32, (+, or4, a,b,c,d),(+, or4 ,e,f,g,h))

out = (+, or9,a,b,c,d,e,f,g,h,i) → out = (+, or2_74ls32, (+, or4, a,b,c,d),(+, or5 ,e,f,g,h,i))

out = (+, or10,a,b,c,d,e,f,g,h,i,j) → out = (+, or2_74ls32, (+, or5, a,b,c,d,e),(+, or5 ,f,g,h,i,j))

out = (+, or11,a,b,c,d,e,f,g,h,i,j,k) → out = (+, or2_74ls32, (+, or5, a,b,c,d,e),(+, or6 ,f,g,h,i,j,k))

out = (+, or12,a,b,c,d,e,f,g,h,i,j,k,l) → out = (+, or2_74ls32, (+, or6, a,b,c,d,e,f),(+, or6 ,g,h,i,j,k,l))

out = (+, or13,a,b,c,d,e,f,g,h,i,j,k,l,m) → out = (+, or2_74ls32, (+, or6, a,b,c,d,e,f),(+, or7 ,g,h,i,j,k,l,m))

out = (+, or14,a,b,c,d,e,f,g,h,i,j,k,l,m,n) → out = (+, or2_74ls32,(+,or7, a,b,c,d,e,f,g),(+,or7,h,i,j,k,l,m,n))

out = (+, or15,a,b,c,d,e,f,g,h,i,j,k,l,m,n,o) → out = (+, or2_74ls32,(+,or7, a,b,c,d,e,f,g),(+,or8,h,i,j,k,l,m,n,o))

out = (+, or16,a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p) → out = (+,or2_74ls32,(+,or8,a,b,c,d,e,f,g,h),(+,or8,i,j,k,l,m,n,o,p))

out = (+, or17,a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q) → out=(+,or2_74ls32,(+,or8,a,b,c,d,e,f,g,h),(+,or9,i,j,k,l,m,n,o,p,q))

out = (+, or18,a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r) → out = (+,or2_74ls32,(+,or9,a,b,c,d,e,f,g,h,i),(+,or9,j,k,l,m,n,o,p,q,r))

out = (+,or19,a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s) → out=(+,or2_74ls32,(+,or9,a,b,c,d,e,f,g,h,i),(+,or10,j,k,l,m,n,o,p,q,r,s))

out = (+, or20,a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t) →
      out=(+,or2_74ls32,(+,or10,a,b,c,d,e,f,g,h,i,j),(+,or10,k,l,m,n,o,p,q,r,s,t))

out = (+, or21,a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u) →
      out=(+,or2_74ls32,(+,or10,a,b,c,d,e,f,g,h,i,j),(+,or11,k,l,m,n,o,p,q,r,s,t,u))

out = (+,or22,a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v) →
      out=(+,or2_74ls32,(+,or11,a,b,c,d,e,f,g,h,i,j,k),(+,or11,l,m,n,o,p,q,r,s,t,u,v))

out=(+,or23,a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w) →
      out=(+,or2_74ls32,(+,or11,a,b,c,d,e,f,g,h,i,j,k),(+,or12,l,m,n,o,p,q,r,s,t,u,v,w))

out=(+,or24,a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x) →
      out=(+,or2_74ls32,(+,or12,a,b,c,d,e,f,g,h,i,j,k,l),(+,or12,m,n,o,p,q,r,s,t,u,v,w,x))
```

Fig. D.3. OR Gate Technology Mapping Rules

12.2. Timing Optimization Rules

Timing optimization rules are divided into two groups:

1. Topographical Rules

2. Gate Delay Rules

Both of these rule sets are presented below:

12.2.1. **Topographical Rules**

out = (*, and2, x, (+, or2, y, z)) → out = (+, or2, (*, and2, x, y), (*, and2, x, z))

out = (*, and2, (+, or2, y, z), x) → out = (+, or2, (*, and2, x, y), (*, and2, x, z))

out = (+, or2, x, (*, and2, y, z)) → out = (*, and2, (+, or2, x, y), (+, or2, x, z))

out = (+, or2, (*, and2, y, z), x) → out = (*, and2, (+, or2, x, y), (+, or2, x, z))

out = (*, and2, x, (+, or2, x, y)) → out = x

out = (*, and2, x, (+, or2, y, x)) → out = x

out = (*, and2, (+, or2, x, y), x) → out = x

out = (*, and2, (+, or2, y, x), x) → out = x

out = (+, or2, x, (*, and2, x, y)) → out = x

out = (+, or2, x, (*, and2, y, x)) → out = x

out = (+, or2, (*, and2, x, y), x) → out = x

out = (+, or2, (*, and2, y, x), x) → out = x

out = (+, or2, z, (+, or2, x, y)) → out = (+, or2, x, (+, or2, z, y))

out = (+, or2, (+, or2, x, y), z) → out = (+, or2, x, (+, or2, z, y))

out = (+, or2, z, (+, or2, y, x)) → out = (+, or2, x, (+, or2, z, y))

out = (+, or2, (+, or2, y, x), z) → out = (+, or2, x, (+, or2, z, y))

out = (*, and2, z, (*, and2, x, y)) → out = (*, and2, x, (*, and2, z, y))

out = (*, and2, (*, and2, x, y), z) → out = (*, and2, x, (*, and2, z, y))

out = (*, and2, z, (*, and2, y, x)) → out = (*, and2, x, (*, and2, z, y))

out = (*, and2, (*, and2, y, x), z) → out = (*, and2, x, (*, and2, z, y))

out = (*, and2, x, x) → out = x

out = (+, or2, x, x) → out = x

out = (*, and2, (*, and2, a, b), (*, and2, c, (*, and2, d, e))) → out = (*, and2, (*, and2, e, b), (*, and2, c, (*, and2, d, a)))

out = (*, and2, (*, and2, a, b), (*, and2, c, (*, and2, d, e))) → out = (*, and2, (*, and2, a, e), (*, and2, c, (*, and2, d, b)))

out = (*, and2, (*, and2, a, b), (*, and2, c, (*, and2, d, e))) → out = (*, and2, (*, and2, d, b), (*, and2, c, (*, and2, a, e)))

out = (*, and2, (*, and2, a, b), (*, and2, c, (*, and2, d, e))) → out = (*, and2, (*, and2, a, d), (*, and2, c, (*, and2, b, e)))

out = (*, and2, (*, and2, a, b), (*, and2, c, (*, and2, d, e))) → out = (*, and2, (*, and2, d, e), (*, and2, c, (*, and2, a, b)))

out = (*, and2, (*, and2, a, b), (*, and2, (*, and2, d, e), c)) → out = (*, and2, (*, and2, e, b), (*, and2, (*, and2, d, a), c))

out = (*, and2, (*, and2, a, b), (*, and2, (*, and2, d, e), c)) → out = (*, and2, (*, and2, a, e), (*, and2, (*, and2, d, b), c))

out = (*, and2, (*, and2, a, b), (*, and2, (*, and2, d, e), c)) → out = (*, and2, (*, and2, d, b), (*, and2, (*, and2, a, e), c))

out = (*, and2, (*, and2, a, b), (*, and2, (*, and2, d, e), c)) → out = (*, and2, (*, and2, a, d), (*, and2, (*, and2, b, e), c))

out = (*, and2, (*, and2, a, b), (*, and2, (*, and2, d, e), c)) → out = (*, and2, (*, and2, d, e), (*, and2, (*, and2, a, b), c))

out = (*, and2, (*, and2, (*, and2, a, b), c), (*, and2, d, e)) → out = (*, and2, (*, and2, (*, and2, e, b), c), (*, and2, d, a))

out = (*, and2, (*, and2, (*, and2, a, b), c), (*, and2, d, e)) → out = (*, and2, (*, and2, (*, and2, a, e), c), (*, and2, d, b))

out = (*, and2, (*, and2, (*, and2, a, b), c), (*, and2, d, e)) → out = (*, and2, (*, and2, (*, and2, d, b), c), (*, and2, a, e))

out = (*, and2, (*, and2, (*, and2, a, b), c), (*, and2, d, e)) → out = (*, and2, (*, and2, (*, and2, a, d), c), (*, and2, b, e))

out = (*, and2, (*, and2, (*, and2, a, b), c), (*, and2, d, e)) → out = (*, and2, (*, and2, (*, and2, d, e), c), (*, and2, a, b))

out = (*, and2, (*, and2, c, (*, and2, a, b)), (*, and2, d, e)) → out = (*, and2, (*, and2, c, (*, and2, e, b)), (*, and2, d, a))

out = (*, and2, (*, and2, c, (*, and2, a, b)), (*, and2, d, e)) → out = (*, and2, (*, and2, c, (*, and2, a, e)), (*, and2, d, b))

out = (*, and2, (*, and2, c, (*, and2, a, b)), (*, and2, d, e)) → out = (*, and2, (*, and2, c, (*, and2, d, b)), (*, and2, a, e))

out = (*, and2, (*, and2, c, (*, and2, a, b)), (*, and2, d, e)) → out = (*, and2, (*, and2, c, (*, and2, a, d)), (*, and2, b, e))

out = (*, and2, (*, and2, c, (*, and2, a, b)), (*, and2, d, e)) → out = (*, and2, (*, and2, c, (*, and2, d, e)), (*, and2, a, b))

out = (+, or2, (+, or2, a, b), (+, or2, c, (+, or2, d, e))) → out = (+, or2, (+, or2, e, b), (+, or2, c, (+, or2, d, a)))

out = (+, or2, (+, or2, a, b), (+, or2, c, (+, or2, d, e))) → out = (+, or2, (+, or2, a, e), (+, or2, c, (+, or2, d, b)))

out = (+, or2, (+, or2, a, b), (+, or2, c, (+, or2, d, e))) → out = (+, or2, (+, or2, d, b), (+, or2, c, (+, or2, a, e)))

out = (+, or2, (+, or2, a, b), (+, or2, c, (+, or2, d, e))) → out = (+, or2, (+, or2, a, d), (+, or2, c, (+, or2, b, e)))

out = (+, or2, (+, or2, a, b), (+, or2, c, (+, or2, d, e))) → out = (+, or2, (+, or2, d, e), (+, or2, c, (+, or2, a, b)))

out = (+, or2, (+, or2, a, b), (+, or2, (+, or2, d, e), c)) → out = (+, or2, (+, or2, e, b), (+, or2, (+, or2, d, a), c))

out = (+, or2, (+, or2, a, b), (+, or2, (+, or2, d, e), c)) → out = (+, or2, (+, or2, a, e), (+, or2, (+, or2, d, b), c))

out = (+, or2, (+, or2, a, b), (+, or2, (+, or2, d, e), c)) → out = (+, or2, (+, or2, d, b), (+, or2, (+, or2, a, e), c))

out = (+, or2, (+, or2, a, b), (+, or2, (+, or2, d, e), c)) → out = (+, or2, (+, or2, a, d), (+, or2, (+, or2, b, e), c))

out = (+, or2, (+, or2, a, b), (+, or2, (+, or2, d, e), c)) → out = (+, or2, (+, or2, d, e), (+, or2, (+, or2, a, b), c))

out = (+, or2, (+, or2, (+, or2, a, b), c), (+, or2, d, e)) → out = (+, or2, (+, or2, (+, or2, e, b), c), (+, or2, d, a))

out = (+, or2, (+, or2, (+, or2, a, b), c), (+, or2, d, e)) → out = (+, or2, (+, or2, (+, or2, a, e), c), (+, or2, d, b))

out = (+, or2, (+, or2, (+, or2, a, b), c), (+, or2, d, e)) → out = (+, or2, (+, or2, (+, or2, d, b), c), (+, or2, a, e))

out = (+, or2, (+, or2, (+, or2, a, b), c), (+, or2, d, e)) → out = (+, or2, (+, or2, (+, or2, a, d), c), (+, or2, b, e))

out = (+, or2, (+, or2, (+, or2, a, b), c), (+, or2, d, e)) → out = (+, or2, (+, or2, (+, or2, d, e), c), (+, or2, a, b))

out = (+, or2, (+, or2, c, (+, or2, a, b)), (+, or2, d, e)) → out = (+, or2, (+, or2, c, (+, or2, e, b)), (+, or2, d, a))

out = (+, or2, (+, or2, c, (+, or2, a, b)), (+, or2, d, e)) → out = (+, or2, (+, or2, c, (+, or2, a, e)), (+, or2, d, b))

out = (+, or2, (+, or2, c, (+, or2, a, b)), (+, or2, d, e)) → out = (+, or2, (+, or2, c, (+, or2, d, b)), (+, or2, a, e))

out = (+, or2, (+, or2, c, (+, or2, a, b)), (+, or2, d, e)) → out = (+, or2, (+, or2, c, (+, or2, a, d)), (+, or2, b, e))

out = (+, or2, (+, or2, c, (+, or2, a, b)), (+, or2, d, e)) → out = (+, or2, (+, or2, c, (+, or2, d, e)), (+, or2, a, b))

out = (+, or2,(+, or2, a, b),(+, or2, c, d)) → out = (+, or2, (+, or2, (+, or2, a, b), c), d)

out = (+, or2,(+, or2, a, b),(+, or2, c, d)) → out = (+, or2, (+, or2, (+, or2, a, b), d), c)

out = (+, or2,(+, or2, a, b),(+, or2, c, d)) → out = (+, or2, a, (+, or2, b, (+, or2, c, d)))

out = (+, or2,(+, or2, a, b),(+, or2, c, d)) → out = (+, or2, b, (+, or2, a, (+, or2, c, d)))

out = (*, and2,(*, and2, a, b),(*, and2, c, d)) → out = (*, and2, (*, and2, (*, and2, a, b), c), d)

out = (*, and2,(*, and2, a, b),(*, and2, c, d)) → out = (*, and2, (*, and2, (*, and2, a, b), d), c)

out = (*, and2,(*, and2, a, b),(*, and2, c, d)) → out = (*, and2, a, (*, and2, b, (*, and2, c, d)))

out = (*, and2,(*, and2, a, b),(*, and2, c, d)) → out = (*, and2, b, (*, and2, a, (*, and2, c, d)))

12.2.2. Gate Delay Rules

out = (!, inv_74ls04, a) → out = (!, inv_74hc04, a)

out = (!, inv_74hc04, a) → out = (!, inv_74as04, a)

out = (!, inv_74as04, a) → out = (!, inv_74hc04, a)

out = (!, inv_74hc04, a) → out = (!, inv_74ls04, a)

out = (*, and2_74ls08, a, b) → out = (*, and2_74hc08, a, b)

out = (*, and2_74hc08, a, b) → out = (*, and2_74as08, a, b)

out = (*, and2_74as08, a, b) → out = (*, and2_74hc08, a, b)

out = (*, and2_74hc08, a, b) → out = (*, and2_74ls08, a, b)

out = (+, or2_74ls32, a, b) → out = (+, or2_74hc32, a, b)

out = (+, or2_74hc32, a, b) → out = (+, or2_74as32, a, b)

out = (+, or2_74as32, a, b) → out = (+, or2_74hc32, a, b)

out = (+, or2_74hc32, a, b) → out = (+, or2_74ls32, a, b)

13. Appendix E, Input to ADACC for VME bus Arbitrator

This appendix presents the user's input file used to describe the original FSM of the VME bus arbiter circuit synthesized in Chapter 6.

```
DEFAULT_TT : 240 ;

INPUT: br1 ;
INPUT: br2 ;
INPUT: br3 ;
INPUT: bbsy;
INPUT: reset;

STATE :
      NAME : s0;
      OUTPUT : bclr = 0;
      OUTPUT : bg1in = 0;
      OUTPUT : bg2in = 0;
      OUTPUT : bg3in = 0;
END : STATE;

STATE :
      NAME : s1;
      OUTPUT : bclr = 0;
      OUTPUT : bg1in = 1;
      OUTPUT : bg2in = 0;
      OUTPUT : bg3in = 0;
END : STATE;

STATE :
      NAME : s2;
      OUTPUT : bclr = 0;
      OUTPUT : bg1in = 0;
      OUTPUT : bg2in = 0;
      OUTPUT : bg3in = 0;
END : STATE;

STATE :
      NAME : s3;
      OUTPUT : bclr = 0;
      OUTPUT : bg1in = 0;
      OUTPUT : bg2in = 1;
      OUTPUT : bg3in = 0;
END : STATE;


STATE :
      NAME : s4;
      OUTPUT : bclr = 0;
      OUTPUT : bg1in = 0;
      OUTPUT : bg2in = 0;
      OUTPUT : bg3in = 0;
END : STATE;
```

```
STATE :
      NAME : s5;
      OUTPUT : bclr = 1;
      OUTPUT : bg1in = 0;
      OUTPUT : bg2in = 1;
      OUTPUT : bg3in = 0;
END : STATE;

STATE :
      NAME : s6;
      OUTPUT : bclr = 1;
      OUTPUT : bg1in = 0;
      OUTPUT : bg2in = 0;
      OUTPUT : bg3in = 0;
END : STATE;

STATE :
      NAME : s7;
      OUTPUT : bclr = 0;
      OUTPUT : bg1in = 0;
      OUTPUT : bg2in = 0;
      OUTPUT : bg3in = 1;
END : STATE;

STATE :
      NAME : s8;
      OUTPUT : bclr = 0;
      OUTPUT : bg1in = 0;
      OUTPUT : bg2in = 0;
      OUTPUT : bg3in = 0;
END : STATE;

STATE :
      NAME : s9;
      OUTPUT : bclr = 1;
      OUTPUT : bg1in = 0;
      OUTPUT : bg2in = 0;
      OUTPUT : bg3in = 1;
END : STATE;

STATE :
      NAME : s10;
      OUTPUT : bclr = 1;
      OUTPUT : bg1in = 0;
      OUTPUT : bg2in = 0;
      OUTPUT : bg3in = 0;
END : STATE;

STATE :
      NAME : s11;
      OUTPUT : bclr = 0;
      OUTPUT : bg1in = 0;
      OUTPUT : bg2in = 0;
      OUTPUT : bg3in = 0;
END : STATE;

STATE :
      NAME : s12;
```

```
    OUTPUT : bclr = 0;
    OUTPUT : bg1in = 0;
    OUTPUT : bg2in = 0;
    OUTPUT : bg3in = 0;
END : STATE;

STATE :
    NAME : s13;
    OUTPUT : bclr = 0;
    OUTPUT : bg1in = 0;
    OUTPUT : bg2in = 0;
    OUTPUT : bg3in = 0;
END : STATE;

STATE :
    NAME : s14;
    OUTPUT : bclr = 0;
    OUTPUT : bg1in = 0;
    OUTPUT : bg2in = 0;
    OUTPUT : bg3in = 0;
END : STATE;

ARC : FROM : s0; TO : s1; ON : br1 * !reset; END : ARC;
ARC : FROM : s0; TO : s14; ON : !br1*!reset*br2 # !br1*!reset*br3; END : ARC;
ARC : FROM : s14; TO : s3; ON : br2 * !reset; END : ARC;
ARC : FROM : s14; TO : s7; ON : !br2 * br3 * !reset; END : ARC;
ARC : FROM : s14; TO : s0; ON : reset; END : ARC;

ARC : FROM : s1; TO : s2; ON : bbsy # reset; END : ARC;
ARC : FROM : s2; TO : s13; ON : !bbsy # reset; END : ARC;

ARC : FROM : s3; TO : s4; ON : bbsy * !br1 # reset; END : ARC;
ARC : FROM : s3; TO : s5; ON : br1 * !reset; END : ARC;

ARC : FROM : s4; TO : s6; ON : br1 * !reset; END : ARC;
ARC : FROM : s4; TO : s11; ON : !bbsy * !br1 # reset; END : ARC;
ARC : FROM : s5; TO : s6; ON : bbsy # reset; END : ARC;
ARC : FROM : s6; TO : s11; ON : !bbsy # reset; END : ARC;

ARC : FROM : s11; TO : s13; ON : 1; END : ARC;

ARC : FROM : s7; TO : s8; ON : bbsy * !br1 * !br2 # reset; END : ARC;
ARC : FROM : s7; TO : s9; ON : br1 * !reset # br2 * !reset; END : ARC;

ARC : FROM : s8; TO : s10; ON : br1 * !reset # br2 * !reset; END : ARC;
ARC : FROM : s8; TO : s12; ON : !bbsy * !br1 * !br2 # reset; END : ARC;
ARC : FROM : s9; TO : s10; ON : bbsy # reset; END : ARC;
ARC : FROM : s10; TO : s12; ON : !bbsy # reset; END : ARC;

ARC : FROM : s12; TO : s13; ON : 1; END : ARC;
ARC : FROM : s13; TO : s0; ON : 1; END : ARC;
END : PARSE;
```